Interner Bericht

DLR-IB-FT-BS-2022-205

From the Cloud to the Clouds: Taking Integrated Modular Avionics on a New Level with Cloud-Native Technologies

Hochschulschrift

Christian Rebischke
Deutsches Zentrum für Luft- und Raumfahrt

Institut für Flugsystemtechnik
Braunschweig

Deutsches Zentrum
für Luft- und Raumfahrt

Institutsbericht
DLR-IB-FT-BS-2022-205

From the Cloud to the Clouds: Taking Integrated Modular Avionics on a New Level with Cloud-Native Technologies

Christian Rebischke

Institut für Flugsystemtechnik
Braunschweig

90	Seiten
22	Abbildungen
0	Tabellen
156	Referenzen

Deutsches Zentrum für Luft- und Raumfahrt e.V.
Institut für Flugsystemtechnik
Abteilung Sichere Systeme & Systems Engineering

Stufe der Zugänglichkeit: I, Allgemein zugänglich: Der Interne Bericht wird elektronisch ohne Einschränkungen in ELIB abgelegt.

Braunschweig, den 20.08.2024

[redacted]

Institutsleitung:	Prof. Dr.-Ing. S. Levedag
Abteilungsleitung:	Dipl.-Ing A. Bierig
Betreuer:in:	M. Sc. W. Zaeske
Verfasser:in:	M. Sc. C. Rebischke

From the Cloud to the Clouds: Taking Integrated Modular Avionics on a New Level with Cloud-Native Technologies

Christian Rebischke
Clausthal University of Technology
[redacted]

March 25, 2022

Ich widme diese Arbeit meinem Vater. Ich bin vielleicht kein guter Musiker geworden, aber ich hoffe wenigstens ein guter Ingenieur geworden zu sein.

Veröffentlichung

Hiermit erkläre ich mich damit einverstanden, dass meine Abschlussarbeit in der Instituts- und/oder der Universitätsbibliothek ausgelegt und zur Einsichtnahme aufbewahrt werden darf.

Göttingen, March 25, 2022

Christian Rebischke

Publication

I hereby agree that my thesis may be displayed in the institute and/or university library and may be retained for inspection.

Göttingen, March 25, 2022

Christian Rebischke

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und nur unter Zuhilfenahme der ausgewiesenen Hilfsmittel angefertigt habe. Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach anderen gedruckten oder im Internet verfügbaren Werken entnommen sind, habe ich durch genaue Quellenangaben kenntlich gemacht. Außerdem wurde diese Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsstelle im Sinne von §11 Absatz 5 lit. b) der allgemeinen Prüfungsordnung vorgelegt.

Göttingen, March 25, 2022

Christian Rebischke

Statutory Declaration

This master thesis is submitted in partial fulfilment of the requirements for the Clausthal University of Technology. I hereby declare that this dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and acknowledgements. The contributions of any other supervisors to this thesis are made with specific reference.

Goettingen, March 25, 2022

Christian Rebischke

Zusammenfassung

Ziel dieser wissenschaftlichen Arbeit ist es Transferwissen aus dem Cloud Computing Bereich auf die Avionik anzuwenden und dazu zu einem heterogeneren Forschungsbild beizutragen. Im Fokus der Arbeit liegt insbesondere der Weg der Avionik Architekturen von einem föderierten System hin zu einem integrierten System, sowie dessen zukünftige Weiterentwicklung. Dabei sollen die Herausforderungen und Lösungen von bekannten Architekturen analysiert und mit neuen Errungenschaften aus dem Cloud Computing Bereich verglichen werden. Insbesondere der Service Orientierted Architecture (SOA) Ansatz spielt in diesem Vergleich eine Rolle, sowie dessen zuverlässige, sichere und kostengünstige Einsatzmöglichkeiten in Flugzeugen, Drohnen oder Raumschiffen. Die Masterarbeit ist wie folgt gegliedert: In der Einleitung wird die Einordnung der Arbeit wiederholt und in einen Zusammenhang mit der Gegenwart gestellt. Im Zweiten Kapitel wird dann der Weg von einer föderierten Avionik Architektur zu einem integrierten System beleuchtet und dessen Probleme, Herausforderungen und Ideen isoliert. Diese gewonnenen Informationen werden nachfolgend aktuellen Cloud-Native Technologien gegenüber gestellt und potentielle Lösungen vorgeschlagen.

Abstract

The goal of this scientific work is to apply transfer knowledge from the cloud computing area to avionics and to contribute to a more heterogeneous research picture. The focus of this work lies in particular in the transformation of avionics architectures from a federated system to an integrated system, as well as its future development. The challenges and solutions of known architectures will be analyzed and compared with new achievements in cloud computing. In particular the Service Orientated Architecture (SOA) approach plays a role in this comparison, as well as its reliable, secure and cost-effective deployment in airplanes, drones or spaceships. The master thesis is structured as follows: In the introduction, the classification of the thesis is repeated and put in context with the current state of the art. Then, in the second chapter, the path from a federated avionics architecture to an integrated system will be shown and its problems, challenges and ideas will get isolated. This gained information is subsequently being compared with current Cloud-Native technologies and potential solutions for these subjects will be proposed.

Contents

Chapter 1

Introduction

The number of performed flights by global airline industries increased from 23.8 million flights (2004) to 38.9 million flights (2019)[95]. This growing number of performed flights puts an enormous pressure on the global aviation industry as a whole. The permanent price pressure lead to demands of cheaper, lighter and smaller flight components[111]. Every inch and every gramm counts in the global business of civil aviation, because every inch less means one possible paying customer more on the plane and every gramm less means less expensive fuel demands for the flight. But it is not only the underlying architecture and the corresponding hardware that plays a big role in the aviation business. The software forming these architectures and running on these devices plays an equal important role in the aviation industry. The development of software is difficult, error-prone, tedious and expensive. This leads to the question why the aviation industry is not exploiting resources and development processes from other industry branches. The open source software movement provides a staggering amount of different technologies for solving problems that are not too different to the problems from the aviation industry. Reasons for this development paralysis are regulation and certification. The civil aviation sector is strictly regulated, thus experimenting with alternatives is expensive and difficult. Furthermore, the existing certification companies are not known for their disruptive technology announcements. Nevertheless this thesis tries to explore a few of these alternatives and tries to suggest topics that might be interesting for further research. Hopefully, it will help justifying further research in this area and incite changes in the inflexible regulation and certification chain. The United States (US) military sector and the US space industry seem to be more willing to experiment with new or existing open source software. For example, the private US space company *SpaceX* had tremendous success with Linux as operating system on their *Dragon* spacecraft[52] and Linux is not only being used by SpaceX[77]. Of course this success is only possible, because the space industry is much more isolated and kept secret than the civil aviation industry with their international standards and guidelines. One of these standards is the DO-178B certification and its successor DO-178C from the Radio Technical

Commission for Aeronautics (RTCA). This certification has strict requirements on flight operating systems. A few of these requirements are real-time capabilities and a transparent and documented development process with design decisions and other documents. While real-time capabilities can be easily added via soft patches (*SpaceX* is exactly doing this with their *Dragon* spacecraft[52]), the documentation and development process seems to be an invincible obstacle for a successful certification process. Another prominent example of open source adoption is the operation of the Cloud Native container orchestration engine Kubernetes in military fighter planes, like the US military plane *U-2*[139]. Unfortunately there is no research on that topic. This thesis tries to change this as well and tries to connect existing research in both areas for creating synergies between them, but for connecting these two areas we need to understand both of these areas first. Therefore, the next chapter will give an introduction to the history of software architectures on planes and will highlight the most important challenges.

Chapter 2

Related Work

2.1 Federated Avionics

To understand the background of this thesis better it is recommended to understand the journey of flight system architectures. Around the 1970's avionics systems evolved from traditional point-to-point wiring to a standard data bus with a federated system architecture[156]. This federated system has been implemented as distributed collection of dedicated computing resources consisting of Line-Replaceable Units (LRUs) or Line-Replaceable Modules (LRMs)[150]. LRUs and LRMs are modular components that are specifically designed for predetermined tasks, such as interacting with certain flight sensors/effectors[76]. Sensors are reading data and effectors are executing certain actions, for example moving the flight gears. The main advantages of LRUs are their atomic behavior and their strict and easily certifiable system design. Each LRU or LRM contains one specific avionic workload and its required computing resources (processors, input and output (I/O) modules, main memory, hard disks and network cards). Figure 2.1 shows a simplified model of the federated architecture with distributed LRUs, sensors, effectors, and a global data bus connecting the components. It visualizes the enormous effort and the huge amount of cables. Duplicating the systems achieves service redundancy and ensures system reliability[111], for the price of duplicating the LRU as a whole. This does not only mean a duplicated hosted function (the actually functionality of the LRU), it also means double as much cables, processors, main memory, network cards and connectors. Weight and complexity disadvantages are not the only problems with the federated architecture approach. Having a dedicated hardware stack for each LRU means not fully saturated potential. Due to safety reasons the LRU will very unlikely use all of its resources. This means there will be always a spare amount of main memory, hard disk or network saturation. Added up over the whole federation architecture this means a huge amount of unused resource potential and unnecessary energy consumption that could be used for other functionality. The task-specific development of LRUs leads to problems

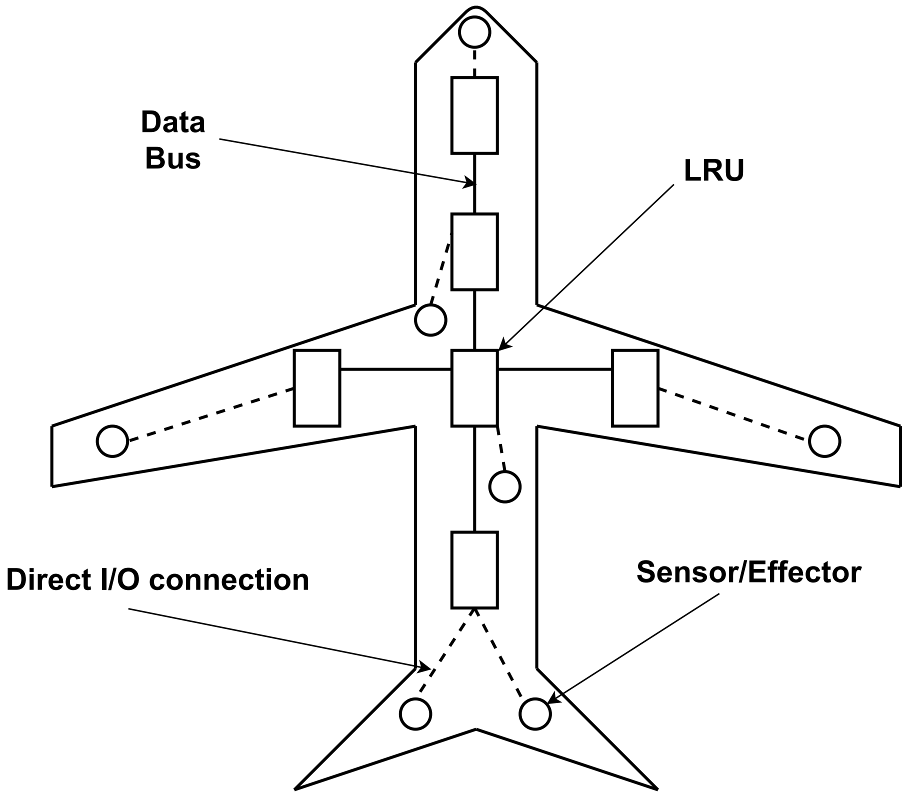


Figure 2.1: Simplified visualization of the federated avionics architecture, showing LRU, sensors, effectors, and the global data bus

with functionality extensions, meaning that LRUs are not easily upgradable or extendable and therefore adjustable to new tasks or functionality. Moreover, in the past, the development of federated architecture components has been closed source and very vendor specific. Specifications for federated architecture components were mostly hidden behind a paywall or non disclosure agreements leading to a decreased developer efficiency and less market competition due to monopolism. LRUs are not easily exchangable between vendors. Figure 2.2 displays the internal structure of the LRU and its interaction with other LRU. Each LRU provides its own hardware stack and hosts exactly one avionic function. It is possible for one LRU to speak with multiple sensors or effectors, but this does not change their core purpose of hosting exactly one function. In section 2.2 this thesis will investigate the next step in the evolution of flight system architectures and how the disadvantages of federated architectures can be fixed.

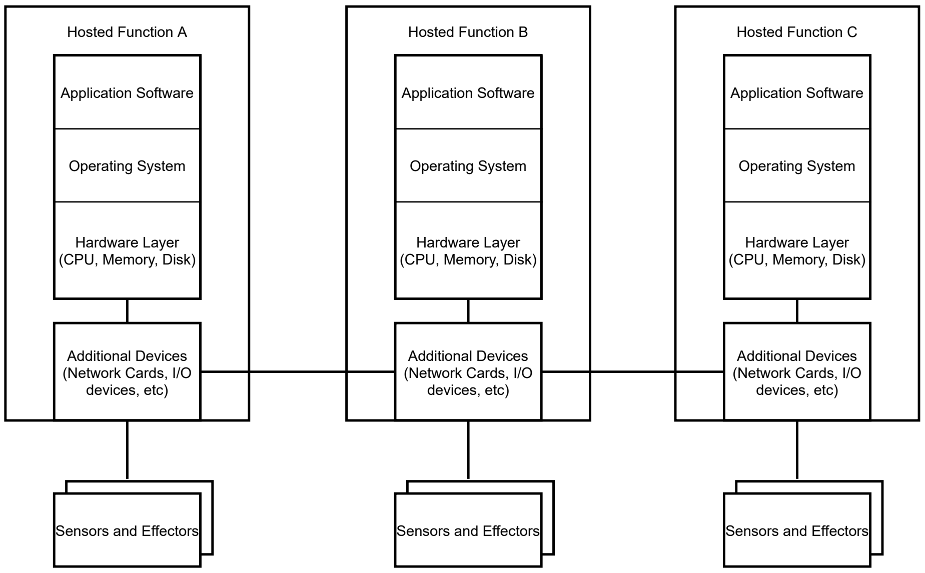

Hosted Function A
Application Software
Operating System
Hardware Layer (CPU, Memory, Disk)
Additional Devices (Network Cards, I/O devices, etc)
Sensors and Effectors
Hosted Function B
Application Software
Operating System
Hardware Layer (CPU, Memory, Disk)
Additional Devices (Network Cards, I/O devices, etc)
Sensors and Effectors
Hosted Function C
Application Software
Operating System
Hardware Layer (CPU, Memory, Disk)
Additional Devices (Network Cards, I/O devices, etc)
Sensors and Effectors

Figure 2.2: Internal system architecture and interaction between LRUs

2.2 Integrated Modular Avionics

Integrated Modular Avionics (IMA) is the direct successor of the federated avionics architecture. The idea behind Integrated Modular Avionics (IMA) is to consolidate the distributed hardware in one central flight cabinet. A Flight cabinet is very similar to a rack in a datacenter. They can host multiple processing units, each comes with its own hardware stack consisting of a Central Processing Unit (CPU), Random Access Memory (RAM), disk space and connectors for input and output[111]. These servers are then plugged into the flight cabinet. Each server can host more than one avionic function and each function is allocated on partitions. Partitions can be created on different ways and has been standardized in ARINC 653 standard[144]. The most common approach is the use of a hypervisor (how this is implemented is being discussed in section 2.4). The flight cabinet provides power and required network connection to the plane's global data bus as described in ARINC 629 standard[59] or ARINC 429 standard[48]. ARINC 629 is the successor of the data bus ARINC 429. Effectors and sensors communicate with the flight cabinet over ARINC 629[111]. Sensors or effectors that are incompatible with ARINC 629 may communicate over remote data concentrators. Remote data concentrators are gathering data from sensors or sending data to effectors over traditional I/O connections. The gathered data or the received actions are communicated via ARINC 629 or ARINC 429. Therefore, remote data concentrators act as bridges between such devices and the data bus. Using a central flight cabinet cannot replace all LRUs in the plane[150]. These LRUs needs to be either integrated into the flight cabinet or connected to the global data bus, for example via remote data con-

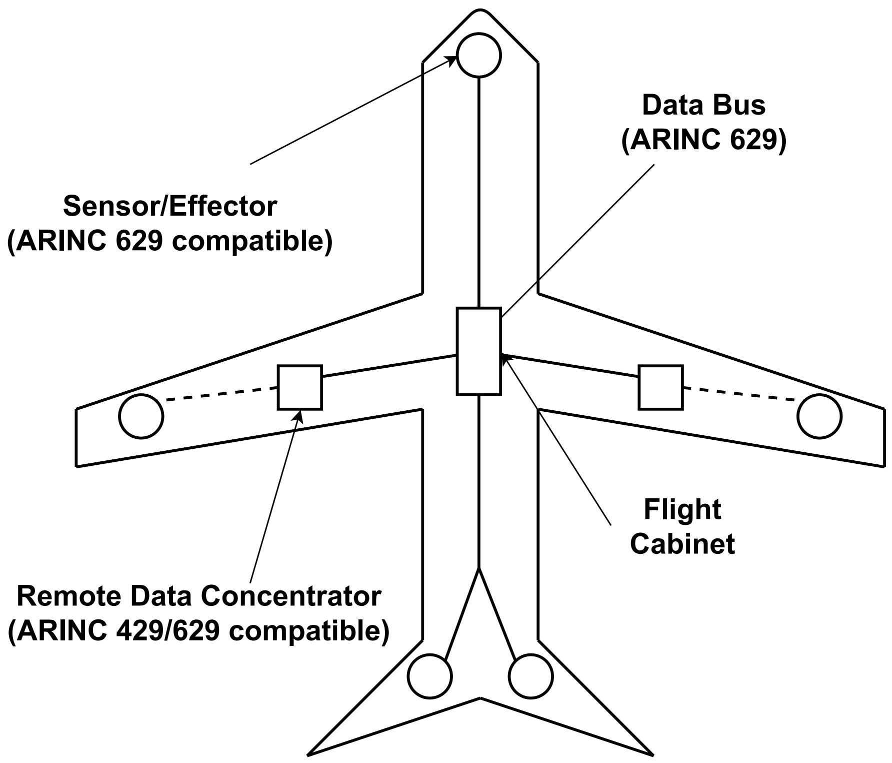


Figure 2.3: Simplified visualization of the integrated modular avionics architecture, showing sensors, effectors, cabinets and the data bus LRUs

centrators. Figure 2.3 depicts a simplified view on integrated modular avionics architecture in a plane. The number of LRUs has been reduced and one central flight cabinet has been introduced. Remote data concentrators are working as bridges between sensors and effectors incompatible with the data bus standard and the hosted functions in the flight cabinet. Figure 2.4 shows the modules inside of such a flight cabinet. The flight cabinet possesses multiple processing units. Each processing unit is comparable to a dedicated computer with its own hardware and operating system. These processing units are being connected via a network layer and each processing unit hosts one or more hosted functions. Hosted functions are isolated from each other and have a fixed predetermined set of resources and execution time.

This system architecture has numerous advantages over the federated avionics architecture. Due to the more centralized approach IMA is able to reduce weight via better cable management and less distributed processing units. Computing resources can be used more efficiently via hosting multiple avionics functions on one processing unit. This leads to a higher system saturation. A positive side effect is less energy consumption and a smaller ecological foot-

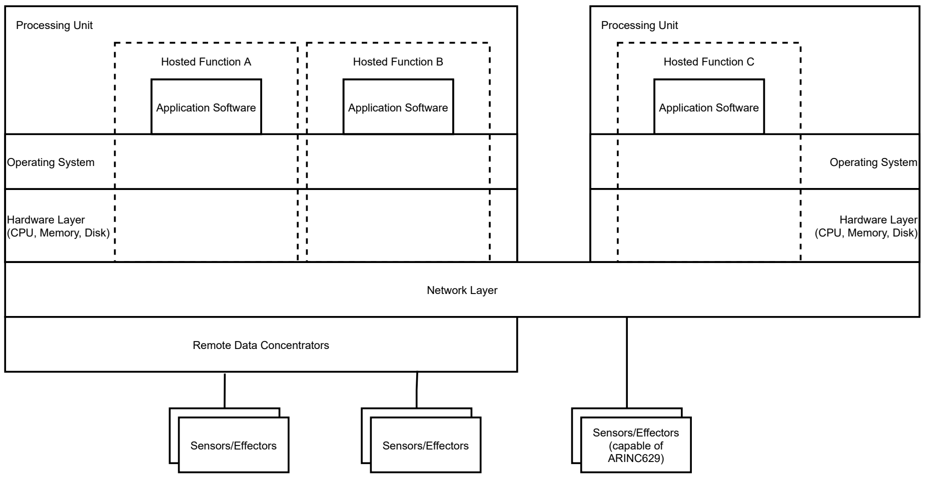


Figure 2.4: View into a flight cabinet

print. The reduced weight creates more space for cargo, fuel or passengers. Hardware consolidation leads to a consolidation of development efforts, which achieves cost and time savings[150]. The common processor allows the developer to focus on the hosted avionic function, enabling a better development experience and less error-prone flight software. Separating software and hardware is a benefit during the certification process, because the certifying authority can certify software and hardware separately. This has just another huge impact on cost and time savings. Additionally, upgrading the hosted function becomes a lot easier, because of the hardware and software separation and less expensive hardware, due to standardized and more common processing units. Another important benefit can be achieved via adopting the idea of open, software and hardware. An open system architecture with open standards can lead to a more competitive market due to industry-wide participation and exchangeability between hardware or software applications. This way development and hardware costs can be reduced, because development cost gets distributed among all contributing companies and mass production of standardized and open hardware shrinks marginal costs[6]. Moreover, the decoupling between hardware and software can have a positive effect on new emerging companies, considering the lower costs[150]. Software virtualization makes it easier to develop the software, without buying expensive physical development kits. The software is being virtualized, tested and can be much later evaluated on real hardware, speeding up the development process and time to market.

2.3 Distributed Integrated Modular Avionics

While IMA introduced a central architecture via consolidating computing resources into one central flight cabinet Distributed Integrated Modular Avionics

(DIMA) takes a different approach. IMA has shown that it is able to successfully reduce the number of components in the plane with increasing number of hosted functions, because of its shared hosting infrastructure[49]. Although this had positive effects on weight and cost management, there is room to improve in form of cable management. Due to the centralized architecture it is necessary to connect every sensor and effector in the plane with the central flight cabinet in the fuselage[83]. This possible increase of cable length can be prevented via using ideas from the federated avionics and IMA. DIMA suggests a distribution of processing units, while taking into account the advantages of a central flight cabinet. Instead of one central flight cabinet it is possible to redistribute the avionics functions over multiple flight cabinets distributed in the plane's fuselage (see also Figure 2.5). This way it is possible to successfully exploit the advantages of the integrated architecture, while achieving further improvements in the cable management[78]. Cable management is not the only possible adjustment for improvement. The commercial cloud computing industry experienced a huge success over the last couple of years and these achievements can be used for creating synergies between these two areas and applying cloud computing ideas in DIMA. According to the National Institute of Standards and Technology (NIST) cloud computing is defined as:

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."[84]

Although not all aspects of cloud computing are applicable on aviation, some of these aspects are. One of these aspects is the separation into different service layers:

- Software as a Service (SaaS)
- Platform as a Service (PaaS)
- Infrastructure as a Service (IaaS)

SaaS is handling all applications. PaaS is responsible for the platform, where these applications are running on (for example a hypervisor or services like webserver or databases). IaaS is the underlying infrastructure (server, storages, networking). These three layers can be mirrored on the aviation world as follows:

- SaaS: Hosted functions
- PaaS: The virtualized processing units or separation kernels
- IaaS: flight cabinets, remote data concentrators, the plane's data bus

Via separating each of these layers the aviation industry gains stronger standardization, more reliability and more effective development workflows. Instead

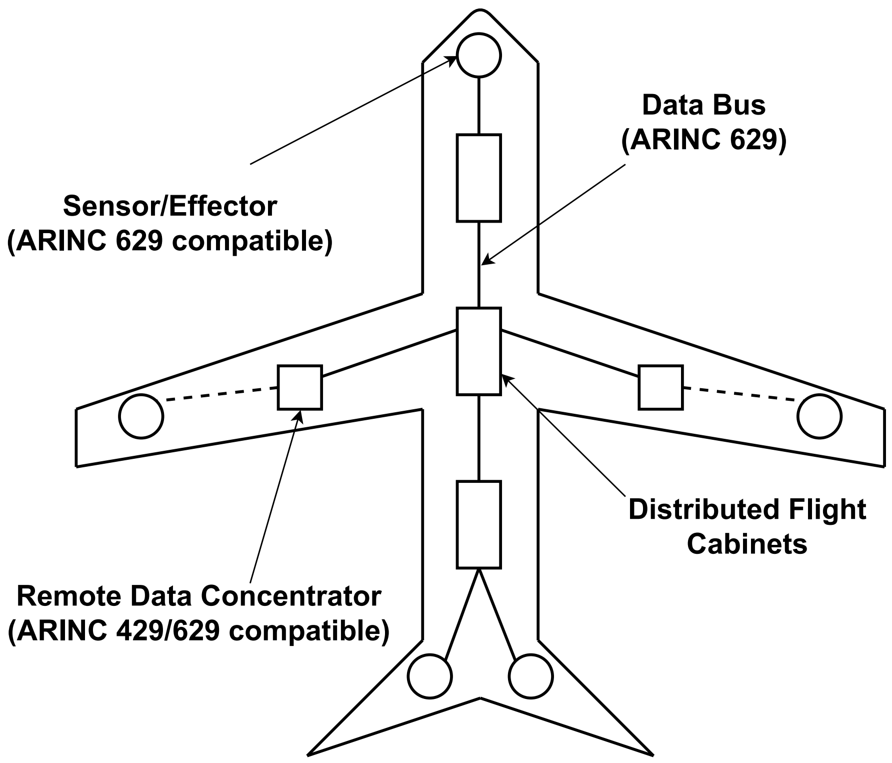


Figure 2.5: Visualization of distributed flight cabinets

of selling one big monolithic system, separation makes it possible to develop products for specific layers and interconnect them with the other layer via standards. These standards can be, as shown in section 2.2, drivers for competition and exchangeability. Another aspect of cloud computing is the free and configurable allocation of applications[78]. Dedicated storage, processor or sensor clusters would allow on-demand access from applications. Nevertheless it is a challenge to make this access reliable and safe. Furthermore, using standardized software may allow interconnection between the plane and ground units (for example: real-time weather data transmitted from the ground to the plane). This is partly comparable to the tactical data link of military units that get real-time radar data on their Horizontal Situation Display (HSD).

2.4 Workload Partitioning Strategies

Planes are composed of complex distributed systems with different tasks, requirements and environmental influences. Every system has its own set of risks and possible impacts. These different risks are known as Design Assurance Level (DAL) or Item Development Assurance Level (IDAL) and have been defined in the Aerospace Recommended Practice (ARP) 4754 as follows[3]:

A Catastrophic system failures

B Hazardous system failures

C Major system failures

D Minor system failures

E No safety effect

Due to these different risk levels it becomes important to isolate systems from each other. A system with a low risk level should never have direct nor indirect impact on another system. In conclusion, the following measures are necessary to satisfy the safety and security requirements in planes:

- Reserved CPU time
- Reserved memory
- Reserved disk space
- Quality of Service (QoS)
- Network isolation
- Process isolation
- Privilege dropping

Reserved CPU time means that every task has a strict time partition to work with. Normally, CPU time is being shared on single processor systems. The operating systems simulates parallelism via fast context switches. Multiple tasks are being executed consecutively. This creates the impression that the system behaves parallel, but in reality it does not. On multi processor systems real parallelism is possible. Reserved memory refers to a fixed memory partition in the RAM. Reserved disk space refers to guaranteed space on a hard disk for storing persistent data. Network isolation and Process isolation ensures a noise free and secure runtime environment. Processes should not be disturbed by other processes either on process or network level. Furthermore, each process has to drop privileges. Dropping privileges increases the safety and security of a service, because it is less likely to interfere with other processes if the process has no privileges to do so. QoS gives certain processes with higher importance or risk level a higher priority. For example, a system that controls the radar should have a higher priority than a system that just controls the ambient lights in the passenger cabin. Many of the above mentioned measures reveal other measures as direct dependencies. For example, it is not possible to do proper QoS without an observable system that gets properly monitored. Another example is the importance of security. On the first view, security might be untangled of safety, but in reality safety and security have a direct effect on each other. If the system is insecure it cannot be safe and reliable, because every possible security incident could undermine all reliability promises. The same applies to safety. An unsafe and unstable system cannot be considered secure, because these weak points in the reliability might weaken the security. Just imagine a system that controls permissions or security related functions. The system itself may be secure, but if it is unreliable it may has a direct effect on the security of other systems that depend on this particular system. Additionally, planes have further constraints and requirements on software. One of these requirements is a guarantee on response. Certain software systems in planes must respond before a given deadline. This is called real-time communication. Real-time communication breaks down into the following types[155]:

Hard The system has to satisfy all constraints. Responses are always on time. If a response would come too late it would inflict damage.

Firm Executed tasks are worthless when their response does not satisfy the time constraint. No damage happens.

Soft There are time constraints for requests and responses, but a tolerance exist. Delays are acceptable (as long as they fit in a specific time frame).

With all of these requirements it is obvious that there need to be technical implementations to satisfy all constraints. The real-time constraints can be satisified via implementing real-time capabilities on operating system or kernel level. Prominent examples are VxWorks' real-time operating system[149], FreeRTOS[114] or the real-time patches for the linux kernel[137]. All other requirements are being solved either via virtualization or via containerization. There are two types of virtualization: Full virtualization and paravirtualization. While full virtualization creates an almost full virtualized hardware including virtualized memory or I/O devices, the paravirtualization does only virtualize the software layer. The hardware stack will not be virtualized. Although this approach increases performance it is considered as less secure. With virtualization it is possible to successfully isolate processes from each other and provide a dedicated workload partition for a specific software with its own RAM and CPU. The key element in this approach is the hypervisor. The hypervisor is the abstraction layer between the hardware, the software and their virtualized counterpart. Type 1 hypervisors run directly on the hardware, while type 2 hypervisors are running on a dedicated host OS. Both hypervisors can usually virtualize a finite number of guests (finite, because the hardware resources are finite). Guests can be either applications or whole operating systems. Many hardware provide direct support for virtualization, such as special instruction sets for Central Processing Units. Containerization uses operating system or kernel level features to isolate processes or resources. In the Linux kernel this usually works via making use of namespaces, kernel capabilities, control groups, union file systems and additional kernel features such like the Extended Berkeley Packet Filter (eBPF). Linux namespaces have been heavily influenced by namespaces in Plan 9 (an alternative operating system by Bell Labs) and zones in Solaris. Contrary to traditional virtualization, namespaces do not need a hypervisor layer, instead the kernel offers a single system call called *setns()*[82]. The Linux namespace Application Programming Interface (API) consists of six dedicated namespaces, each responsible for a different isolation environment[91]:

mnt responsible for filesystem mount points

pid responsible for processes

net responsible for the network stack

ipc responsible for Inter-Process Communication (IPC)

uts responsible for Unix Time Sharing (UTS) and setting hostname information

user responsible for User IDs (UIDs)

cgroup responsible for control groups (cgroups)

time responsible for time

The first namespace *mnt* specifies the filesystem hierarchy for the isolated process or a group of processes. With this namespace it is possible to mount only a specific directory structure into the isolated environment. This ensures that a process from environment *A* cannot access files on the host system or in another environment. The *mnt* namespace has become especially useful in combination with union file systems. Union file systems are layered filesystems and allow to mount multiple directory structures over each other. Docker uses this technology for combining different layers to a full Docker container image. The second namespace *pid* gives the system the opportunity to run multiple processes with the same Process IDs (PIDs) on the host system. Running multiple processes with the same PID on one system fulfills different purposes. On one hand this provides reliability across different hosts. Via this approach it is possible to run a process with the same PID in different environments on different physical hosts. On the other hand it provides isolation. The process can only see other processes contained in the same namespace[92]. The *net* namespace is responsible for everything network related. It provides the isolated environment with Internet Protocol (IP) addresses, ports, routing tables and virtual network devices. *net* is mostly being used for isolating network connections from each other. For example a namespace *A* could provide its own ports and internal IP addresses, while a namespace *B* could provide the same port. The ports can be forwarded to the host via port-forwarding. The *ipc* namespace handles everything Inter-Process Communication (IPC) related (message queues, Linux signals like *sigkill* for killing processes etc). For dealing with different hostnames in different namespaces the *uts* namespace is being used. The last three namespaces implemented by Linux are the *user*, *cgroup* and *time* namespaces. *user* is responsible for User IDs (UIDs) and Group IDs (GIDs). With the namespace *user* it is possible to differentiate users or groups with the same UIDs or

GIDs in different environment. For example, having the *root* user (the administrator account on Linux systems) in different isolated environments. The user *root* is identified by the UID 0 and the Group ID (GID) 0. Different times in isolated environments are provided by the *time* namespace. The last namespace, the *cgroup* namespace, is responsible for the control groups (cgroups). A cgroup enables the system to limit and monitor resources inside of a namespace via providing a pseudo-filesystem called cgroupfs[80]. Each cgroup consists of one or more processes and each process is being modified by a subsystem (also known as resource controller). Subsystems work as modifiers for these processes. Via these modifiers it is possible to set specific limits or monitor the processes. For example, limiting the available CPU time or memory for a process, increasing the priority for network traffic, limiting access to I/O devices or monitoring events created by a process[80]. The last two missing building blocks for secure containers are kernel capabilities and the Extended Berkeley Packet Filter (eBPF). Kernel capabilities provide a more granular access to kernel features such like mounting or creating ports[104]. Common kernel capabilities are *CAP_CHOWN* (changing the owner and group of a file), *CAP_NET_BIND_-SERVICE* (allow the binding of ports under 1024), *CAP_SYS_ADMIN* (allow system administrator access), *CAP_NET_ADMIN* (allow network administrator access, a complete list can be found in the corresponding man page for kernel capabilities[81]). The Extended Berkeley Packet Filter (eBPF) is the successor of the Berkeley Packet Filter (BPF). The Berkeley Packet Filter (BPF) is well known for its human readable filtering language that allows filtering, monitoring and analyzing packets in computer networks. Core of the BPF is a just-in-time (JIT) compiler that translates the human readable BPF language into machine readable byte code. Contrary to its predecessor, eBPF is not limited to network packets. With Linux version 3.18, eBPF has been embedded into the Linux kernel as virtual machine and allows filtering for any sort of Inter-Process Communication (IPC), such like socket connections, system calls (syscalls) or kernel permission delegations. Figure 2.6 gives an overview over different workload partitioning and deployment techniques. On the far left is the traditional setup with the hardware, an operating system running on the hardware, the applications and their shared libraries. The second pillar shows a type 2 hypervisor with a hardware, an operating system, shared libraries on the operating system, a hypervisor and virtual machines. Each virtual machine comes with their own operating system, shared libraries and the actual payload (the application with the workload). The type 1 hypervisor simplifies this setup with running a hypervisor directly on the hardware. In this case, the hypervisor is hypervisor and operating system at once. The container deployment comes with the hardware, an operating system and its shared libraries and a container runtime. Each container can consist of different parts. A container can consist of only one static built executable or, an application and its shared libraries. Shared libraries minimize the size of software via sharing common methods, for instance a library for Remote Procedure Calls (RPC) or Domain Name System (DNS) lookups. When the executable has been linked dynamically, the executable can dynamically look up these common methods. Contrary to dynamic linking is

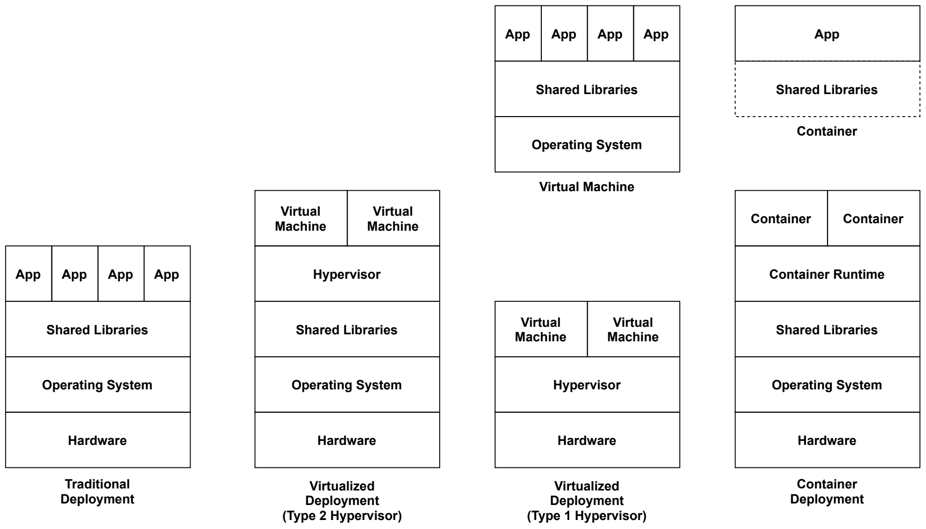


Figure 2.6: Overview over different workload partitioning and deployment techniques

static linking, where a software is being compiled with all its dependencies into one executable. Although all deployment techniques may look similar at a first glimpse, they are indeed completely different. A virtual machine allows stricter isolation compared to running all applications without virtualization and a container can consist of a single binary.

2.5 Challenges and Opportunities

Prior, this thesis has explored past, present and future avionic system architectures. On the next pages it is going to examine challenges in these avionic system architecture designs and possible enhancements in the avionic domain in general. This document is explicitly ignoring the certification process of software. It is known that a certification process in the avionic domain can be slowly and very costly, due to its high risk to passengers, flight personnel and civilians on the ground. Enhancing this certification process via altering the certification itself, while matching the right balance between safety and development speed or costs, is explicitly not topic of this research. Instead, this document tries to focus on few main questions. The first identified challenge is the increase of development speed in the avionic domain. Pushing innovations and new technologies fast is a major success factor. Maintaining a hardware and software stack for a traditional avionic system architecture, like the federated architecture, is a huge cost and time disadvantage. The developer cannot use the standard development tools and needs to adapt to the new environment. Moreover, buying the necessary equipment and constantly adjusting the software to this equipment are toilsome and complicated. Every hour of additional work is a cost factor that leads to higher unit prices, smaller profit margins and less time for more features. Costs directly lead to the next challenge: Decreasing development costs and increasing feature output. Considering the tight entanglement of soft- and hardware in federated systems these stacks become very cost intensive. Also, the feature output is limited to the required hard- and software, hence limiting possible performance or functionality gains. Performance is another challenge. As a result of heavy use of virtual machines the performance gains are not as high as they could be. The hard- and software limitations make the system less extensible and less adjustable. Another challenge is the proprietary avionic system development environment. Proprietary standards, hardware- and software have devastating effects on innovation, safety, security, costs and technological development. Innovation is affected negatively, because of less communication between companies. Open architectures, software or hardware, increase communication and share knowledge between all development entities. This shared knowledge and improved communication is known for increasing security of software[54]. Restricting access to the source code may help restricting the attacker's knowledge for executing attacks, but it is shown that "keeping the source code closed appears to be difficult"[85]. Furthermore, patches for open source software are released twice as fast as for closed source software[154]. The increased pace of patching the software underlines the increased development speed and its connected benefits. Many of the highlighted challenges are not new for the tech industry. The cloud industry has solved several of these challenges successfully.

# Chapter 3

Cloud-Native

3.1 Introduction

Containers are not a new concept. The idea behind them goes back to systems like Solaris, FreeBSD or Linux. FreeBSD has released their first container-alike isolation environment, called *jails*, in 2000[55]. Jails are heavily being influenced by the chroot system call on Unix systems. The chroot system call has been introduced with Version 7 Unix in 1979[13]. The concept has been there all the time so what was missing? Although containers were in use by system administrators the workflow was not developer friendly enough. Big companies like Google realized the potential of containers early and started to heavily use them in their own infrastructure, but the overall success came with open source technologies like Docker or Kubernetes. It turned out that opening up the development process of these technologies has been the perfect foundation for creating a community that spans over several companies and industry branches. This community is under supervision of the Linux Foundation and is called Cloud Native Computing Foundation (CNCF)[94]. The Cloud Native Computing Foundation (CNCF) built an enormous ecosystem for running and orchestrating containers. Figure 3.1 visualizes the scale of the CNCF related projects[22]. Considering that that the CNCF was founded in 2015 this is clearly a huge success for the containerization of applications in different industry branches, especially for the cloud computing industry. Looking at the landscape it can be seen that the CNCF covers different areas and expands wide beyond containerization. Just to name a few different areas and their sub areas on the landscape:

App Definition and Development databases, streaming and messaging, application definition and image build, continuous integration and delivery

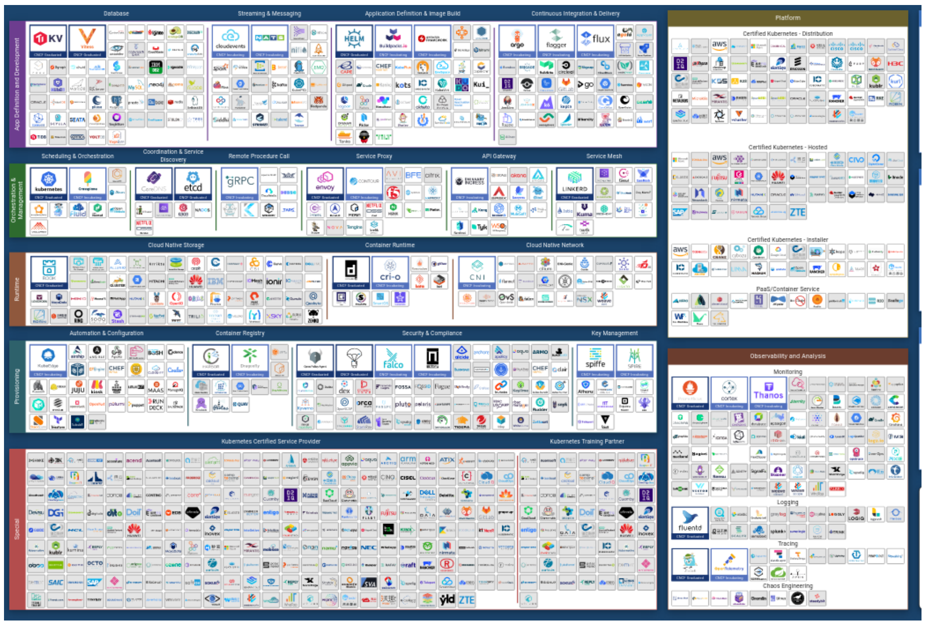
Orchestration and Management scheduling and orchestration, coordination and service discovery, remote procedure call, service proxy, api gateway, service mesh

Runtime Cloud-Native storage, container runtime, Cloud-Native network

Provisioning automation and configuration, container registry, security and compliance, key management

Observability monitoring, logging, tracing, chaos engineering

While some of the above areas play only a minor role in the aviation industry some areas are usable or will play a bigger role in the future. Especially, areas like scheduling and orchestration or container runtime are of interests for the aviation industry, because these areas provide related functionality compared to current solutions in actual use within the aviation industry.



Database
Streaming & Messaging
Application Definition & Image Build
Continuous Integration & Delivery
Platform
Scheduling & Orchestration
Coordination & Service Discovery
Remote Procedure Call
Service Proxy
API Gateway
Service Mesh
Cloud Native Storage
Container Runtime
Cloud Native Network
Automation & Configuration
Container Registry
Security & Compliance
Key Management
Kubernetes Certified Service Provider
Kubernetes Training Partner
Observability and Analysis

Figure 3.1: Cloud-Native Computing Foundation Landscape

3.2 Docker

Docker (released in 2013[43]) has revolutionized the way containers are being used. Before Docker, the primary users of containers have been sysadmins, who used containerization to isolate environments from each other. With Docker this workflow has changed. Docker has introduced a unique file format, called Dockerfile, for describing a container consisting of different layers. These layers are stacked on each other, with the help of union file systems, for a final Docker image. This process has a few advantages. Each layer can be cached separately, leading to less disk usage and faster container build times. Layers do not need to be from the same author. In Docker it is very common to re-use common Docker images as a base layer and add additional payload to them via adding layers on top of the base layer. The base layer is often called the base image and the most popular base image is the *scratch* image. *Scratch* is just an empty Docker layer. All other base images use the *scratch* image as start point for the build process. Base images can be whole Linux distributions like Debian, proprietary operating systems of the industry or just a directory structure. For example, Google provides a *distroless* Docker image that just contains time zone data and the global certificate chain[33]. The *distroless* image is mainly in use for shipping static binaries inside of Docker. This is common practice for languages like Rust or Go. Shipping just the binary leads to a much smaller attack surface. Having no shell inside of the image makes it impossible for an attacker to gain shell access to such a container and missing additional executables make it difficult to exploit additional executables for building an attack chain (for example via using network utilities for opening a reverse shell into the container). Listing 3.1 shows an example multi-stage Dockerfile with different layers. Each line in the file is a separate layer in the final image and each *FROM* statement defines a different stage in the Docker image build process. In the first stage of the Dockerfile a small hello world program will be written in the programming language Go (often referred to as Golang) and compiled. In the second stage, the executable from the first stage will be copied to the new second stage, placing it into a *distroless* image for reducing the possible attack surface and making the executable available for further distribution as Docker image. The first stage starts with the statement *FROM docker.io/golang:1.17 AS GO_BUILD*, setting the base image to *golang* with version 1.17 from the Docker registry *docker.io* and naming this stage *GO_BUILD*. A Docker registry is a public or private repository for Docker images. The second statement *WORKDIR /go/src/app* sets the working directory for the build process. It is comparable to moving to a folder in a Windows system or changing the directory in a Linux system via the command line command *cd*. The next three statements starting with *ENV* are setting environment variables. Environment variables are an operating system feature for setting variables outside of programs for configuration purposes. *GOOS* sets the target operating system to Linux, *GOARCH* sets the architecture to amd64 and *CGO_ENABLED* disables the libc support for the Go build process. Disabling the libc support is equal to disabling linking to shared libraries, resulting to a static compiled binary that can be moved on different systems as long as the target operating system and architecture stays the same. With the next three *RUN* statements it is possible to execute commands in the base image *docker.io/golang:1.17* (this also means that our base image is actually based on Linux). The *echo* command prints our Go hello world code and pipes it into a file called *main.go*. Using *go mod init app* initializes the Go build and dependency system with a name for the app. Running *go build -o /out/app .* will build the *main.go* file in the current directory and move the created executable *app* to the directory */out/*. Using the next *FROM gcr.io/distroless/base* statement a new stage will be build with a new base image for the final image holding the executable and reducing possible attack surface by utilizing Google's *distroless* images. The *COPY* statement will copy the built executable from the first stage into the new second stage and the *ENTRYPOINT* statement will set the path of the application as entrypoint for the container startup. It is important to differ between images and containers. Containers are running instances of images. The entrypoint is the start point from each container and be compared to the main function of most C-alike programming languages or the init system of a modern operating system. Listing 3.2 shows the output of building the Docker image. Each step is equal to one layer of the final image and each layer has a unique Secure Hash Algorithm 2 (SHA256) checksum over the binary content of the layer for ensuring integrity and reproducibility. This minor information has a unique impact on supply chain security, developer experience and costs, because with each layer cryptographically identified it is much more difficult to tamper with the build process, allowing secure access to cached layers and leading to faster development speed and lower costs. The final image can be run as container via executing *docker run* leading to the promised *Hello World!* line.

Listing 3.1: A multi stage Dockerfile with different layers

```
FROM docker.io/golang:1.17 AS GO_BUILD
WORKDIR /go/src/app
ENV GOOS=linux
ENV GOARCH=amd64
ENV CGO_ENABLED=0
RUN echo "package main\nimport \"fmt\"\nfunc main() { fmt
    .Println(\"Hello world!\") }" > main.go
RUN go mod init app
RUN go build -o /out/app .

FROM gcr.io/distroless/base
COPY --from=GO_BUILD /out/app .
ENTRYPOINT ["./app"]
```

Listing 3.2: Output of a docker build run with shortened SHA256 checksums

```
STEP 1: FROM docker.io/golang:1.17 AS GO_BUILD
STEP 2: WORKDIR /go/src/app
--> Using cache baa3b5061ca
--> baa3b5061ca
STEP 3: ENV GOOS=linux
--> Using cache fe142d360ee
--> fe142d360ee
STEP 4: ENV GOARCH=amd64
--> Using cache 49771a6a4a6
--> 49771a6a4a6
STEP 5: ENV CGO_ENABLED=0
--> Using cache edf307d8374
--> edf307d8374
STEP 6: RUN echo "package main\nimport \"fmt\"\nfunc main
    () { fmt.Println(\"Hello world!\") }" > main.go
--> Using cache ca435b8adf9
--> ca435b8adf9
STEP 7: RUN go mod init app
--> Using cache 19b7a23f24d
--> 19b7a23f24d
STEP 8: RUN go build -o /out/app .
--> Using cache baeee09e19f
--> baeee09e19f
STEP 9: FROM gcr.io/distroless/base
STEP 10: COPY --from=GO_BUILD /out/app .
--> Using cache 5ab01f9acf7
--> 5ab01f9acf7
STEP 11: ENTRYPOINT ["./app"]
--> Using cache 288ac81c34b
--> 288ac81c34b
```

3.3 Open Container Initiative

In June 2015 leading container projects, such like Docker and CoreOS (a minimal Linux distribution for running containers), launched the Open Container Initiative (OCI) under auspices of the Linux Foundation[2]. The OCI's purpose is the standardization of Docker's container image specification (image-spec[102]), container runtime (runtime-spec[103]) and content distribution (distribution-spec[96]). These three specifications address everything from building a container image, distributing the image over a container registry, and running the container. Standardization enables interchangeability of all three implementations and increases possibilities for their production usage. In the following, this thesis will have a short glimpse on each of the specifications. A full writeup would break the constraints of this thesis. It is recommended to read the original specification, if more information is needed. The image specification has five essential components:

manifest provides a set of layers and a configuration for a specific image runnable on a specific architecture and operating system

image index provides a set of manifests with different architectures and operating systems

image layout provides a file system layout (the contents of the image)

filesystem layers provides information about merging different layers to one image

configuration provides the configuration of the image, provided by the Dockerfile (environment variables, entrypoints, etc)

Listing 3.3 depicts a very simplified example image manifest. The SHA256 checksums have been shortened for saving space in this document. The manifest shows the schema version, a configuration, multiple layers and annotations. The configurations and layers have media types, a field for the size and a digest (a checksum) of the addressable content. Annotations provide further information for the image. The OCI manifest specification defines a few predefined annotations, but at last these are just customizable strings. Annotations allow additional metadata for images, thus providing information for various other use cases, like easier search in an OCI compatible registry or information about its maintainers. With the manifest it is possible to address the content via its offset and to provide integrity via digests [101]. One or more image manifests can form an image index. An image index is a set of different image manifests for different operating systems and architectures [99]. Listing 3.4 illustrates such an image index with support for two platforms. The first addressed manifest links to a manifest for MacOS with M1 chipset. The second addressed manifest refers to a manifest for Linux with amd64 chipset. Furthermore, it is possible to add additional annotations to the image index definition. As mentioned earlier, an OCI image consists of different layers. These layers are defined in the image layout and the filesystem layers specifications. One or more compressed layers are applied on top of each other to create the entire root filesystem of the image[97]. The filesystem layers specification can handle additions, modifications and removals of various file types, such like directories, sockets, symbolic links, block devices, or regular files[97]. All of these layers are described as an image layout. Each layer is stored in the image layout as Binary Large Object (BLOB) and addressable via its SHA256 digest. An OCI image layout is comparable to a directory in a filesystem. A BLOB directory stores all layers, an index.json file provides platform interoperability via establishing the relationship between each layer and their target platform [100]. OCI layout files work as marker and version identifier of the OCI layout. While the prior mentioned specifications define the structure of the OCI image, the configuration specifies the environments, entrypoints, volumes, working directory, commands that should be executed and other configuration parameters. The *docker image inspect <docker image>* command prints a full OCI image configuration for a given image. Listing 3.5 shows the output of this command for the prior mentioned OCI image, built from Listing 3.1. The output has been cut down to the most important parts. Clearly visible is the environment in form of environment variables, the command, the working directory, an entrypoint, the Docker version, a build date and the root file system layers. If compared with Listing 3.1 the similarities are obvious. This image configuration works as interface for the runtime specification. The Container Runtime Interface (CRI) provides an interface for running OCI compatible images. Common implementations of the CRI are Docker, Containerd[25] or cri-o[30]. Containers are encoded as filesystem bundles. A filesystem bundle consists of the container configuration as specified in the OCI image specification and the container's root filesystem (referenced in the image configuration)[41]. These filesystem bundles are executed by the container runtime. The container runtime manages the state and the lifecycle of the container. Every container runtime implementation must fulfil the CRI specification's lifecycle operations and commands for basic runtime interaction (create, start, stop, kill, etc). Distributing a container image is described by the OCI distribution specification. The distribution specification defines all necessary components of an OCI image compatible registry and its basic operations. Described are the pull of an image (the act of downloading BLOBs and manifests from the registry), the push of an image (the act of uploading BLOBs and manifests to the registry), and the registry itself.

Listing 3.3: An OCI image manifest with two layers, shortened sha256 checksums and two annotations

```
{
    "schemaVersion": 2,
    "config": {
        "mediaType": "application/vnd.oci.image.config.v1
            +json",
        "size": 8652,
        "digest": "sha256:b5bb9d8014a0f..."
    },
    "layers": [
        {
            "mediaType": "application/vnd.oci.image.layer
                .v1.tar+gzip",
            "size": 40270,
            "digest": "sha256:c739918a22764..."
        },
        {
            "mediaType": "application/vnd.oci.image.layer
                .v1.tar+gzip",
            "size": 14743,
            "digest": "sha256:9f767b0b078d2..."
        }
    ],
    "annotations": {
        "org.opencontainers.image.authors": "Christian
            Rebischke",
        "org.opencontainers.image.title": "A very easy
            manifest example"
    }
}
```

Listing 3.4: An OCI image index with multiple manifests, shortened sha256 checksums and platform specifications

```
{
    "schemaVersion": 2,
    "manifests": [
        {
            "mediaType": "application/vnd.oci.image.
               manifest.v1+json",
            "size": 9783,
            "digest": "sha256:6650b51fab815ad7fc331f...",
            "platform": {
                "architecture": "arm64",
                "os": "darwin"
            }
        },
        {
            "mediaType": "application/vnd.oci.image.
               manifest.v1+json",
            "size": 9543,
            "digest": "sha256:ed22e9fb1310cf6c2decab...",
            "platform": {
                "architecture": "amd64",
                "os": "linux"
            }
        }
    ],
    "annotations": {
        "org.opencontainers.image.created": "2021-12-19
           T16:39:57-08:00",
    }
}
```

Listing 3.5: An OCI image configuration created via docker image inspect (shortened)

```
{
    "RepoTags": [
    "hello-world:latest"
    ],
    "Created": "2021-12-19T07:47:29.318765235Z",
    "ContainerConfig": {
    "Hostname": "motoko.shibumi.dev",
    "User": "0",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "Tty": false,
    "Env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/
            usr/bin:/sbin:/bin",
        "SSL_CERT_FILE=/etc/ssl/certs/ca-certificates.crt
            "
    ],
    "Cmd": [
        "/bin/sh",
        "-c",
        "#(nop) ",
        "ENTRYPOINT [\"./app\"]"
    ],
    "Image": "sha256:bc721b69a6af4bb...",
    "WorkingDir": "/",
    "Entrypoint": [
        "./app"
    ],
    "DockerVersion": "20.10.12",
    "Architecture": "amd64",
    "Os": "linux",
    "Size": 22013044,
    "VirtualSize": 22013044,
    "RootFS": {
    "Type": "layers",
    "Layers": [
        "sha256:c0d270ab7e0db0...",
        "sha256:e3d24823466584...",
        "sha256:589f2bccf144b2..."
    ]
    },
}
```

3.4 Container Interfaces

With the rise of the Open Container Initiative (OCI), more and more container management systems and orchestration services have been developed. While the implementations of the Container Runtime Interface (CRI) provides basic runtime functionality for executing containers, it became clear that an interface for running containers is not enough. The usecases for containers became more complex and orchestration services became distributed over large network topologies. This distribution lead to the necessity of standardized networking and storage interfaces. These interfaces are called Container Network Interface (CNI) and Container Storage Interface (CSI). This section summarizes both interfaces and gives a good introduction. However, this section is not meant as complete guide for implementing these interfaces. CSIs provide an interface for new storage providers. A storage provider is the vendor of the plugin implementation[24]. Storage providers can be providers for local or remote storage. Every local storage differs and there are different ways on a system to provide storage for an application. Local Storage can be offered via device mappers, the Logical Volume Manager (LVM), raw block devices, volume groups or many other operating system and hardware dependent systems. Remote storage is equally complicated due to additional network capabilities and protocols, such like Network File System (NFS) or ceph (the distributed, fault-tolerant storage platform). Each of these storage providers have a different API and different ways to read and write data on them. Container orchestrators implement the Container Storage Interface (CSI). Communication between the CSI plugin and the container orchestrator happens over Remote Procedure Calls (RPC). The most common RPC protocol in the cloud-native environment is Google Remote Procedure Calls (gRPC). gRPC (invented in 2016 by Google) has the huge advantage that it is language independent, supports a very simple service definition via Protocol Buffers and bi-directional streaming with several security features, for instance authentication[51]. Protocol Buffers is an Interface Definition Language (IDL). IDLs offer a, for the developer, convenient way to describe an RPC interface in a language independent and human-readable way. gRPC and Protocol Buffers are the tools of choice for the CSI. The CSI presents basic functionality for dynamic provisioning of volumes (on-demand provisioning), attaching or detaching volumes, mounting or unmounting volumes from an orchestration node, creating or deleting snapshots or provisioning new volumes[24]. Snapshots are one-time states of a volume with data, they are not equal to backups. Snapshots allow the storage provider to go back in time and restore data for a given time frame. For this operation the storage provider requires necessary metadata and a running service. This is what distances them from a backup. A backup offers a reliable way to restore a full dataset in case of a disaster (for instance a hardware outage). A snapshot does not offer this reliability and may fail if the underlying system is damaged. Usually, a storage provider implements an Application Programming Interface (API) that is CSI compatible or a third-party implements a service that interprets the APIs of the storage provider plugin and translates it, to be CSI compliant. When container orchestrators like Docker became distributed over multiple nodes the network became more relevant. The Container Network Interface (CNI) specification addresses this problem and proposes a standard interface for enhanced network capabilities for future container runtime and container orchestration solutions. Equally to the CSI specification, the Container Network Interface (CNI) specification is language agnostic, but the reference implementation is written in Go[23]. As of writing this thesis, there are already over 20 CNI compatible plugins from various cloud providers, hardware vendors or third-parties. These plugin differ significantly, because they address different use cases, problems or network environments. The CNI specification abstracts these differences and provides a common interface for all of them. CNI plugins are not bound to one specific layer of the Open Systems Interconnection model (OSI). For instance, the CNI plugin *flannel* focuses on OSI layer 3 only, hence providing every container with its own unique routable IP address via spanning an IPv4 network over all container orchestration nodes in a cluster (a cluster is a group of multiple orchestration nodes, responsible for running containers or managing the cluster)[44]. Other CNI plugins such like *cilium* or *calico* offer support for more OSI layers and more features, such like traffic encryption, traffic monitoring, network policies or more graduated network and routing capabilities.

3.5 Kubernetes

3.5.1 Introduction and History

Although Docker has been released in 2013, Google has been using Containers for a much longer period of time. According to Google engineers, they are managing Linux containers at scale since at least 2006[12]. This long history of Linux container management lead to several internal container orchestration platforms within Google, such like Borg[146], Omega[121] or Kubernetes[74]. While Borg and Omega are internal projects and therefore proprietary, Kubernetes is open source and freely available on GitHub[69]. Since the Kubernetes release in 2014[68], Kubernetes has started a chain reaction in the industry. More and more projects have been founded around Kubernetes as distributed container orchestrator. The Cloud Native Computing Foundation (CNCF) landscape lists a plethora of different projects and project domains around Kubernetes, such like continuous integration and continuous delivery frameworks, service proxies, service meshes, Kubernetes compatible container runtimes, Container Network Interface (CNI) or Container Storage Interface (CSI) plugins or security addons for Kubernetes. This extensibility is one of Kubernetes' many success drivers. Kubernetes offers everything necessary for running OCI compatible containers distributed on one or more Linux hosts, called nodes. These OCI compatible containers do not even need to be containers in the traditional sense. It is possible, with a little bit of effort and the KubeVirt project[117], to run small virtual machines with Kubernetes. Such virtualized containers are called microVMs. However, on default Kubernetes focuses on running containers as abstracted by namespaces, cgroups and other Linux kernel features. Kubernetes does this slightly differently to other container orchestrators. The smallest manageable unit in the Kubernetes world is being called a *pod*. A pod consists of one or more containers and possesses a few additional properties to enhance container management and orchestration. These differences will be explained in more detail in the following sections.

3.5.2 Architecture

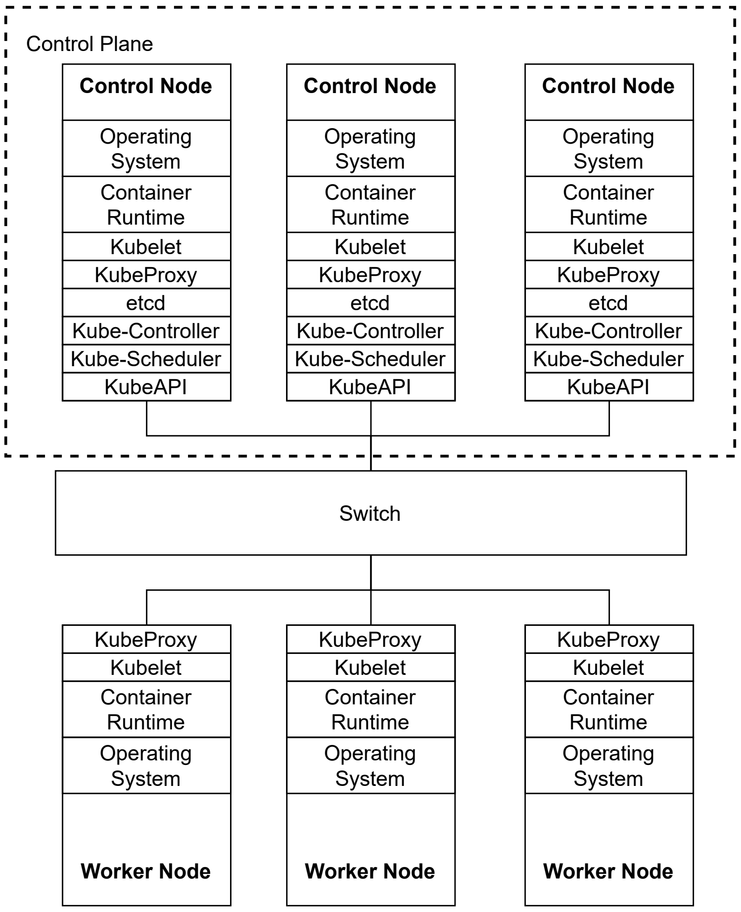


Figure 3.2: Overview over a Kubernetes cluster with three control nodes and three worker nodes

In this section, this thesis explores the architecture of Kubernetes. Figure 3.2 depicts a Kubernetes cluster with three control nodes and three worker nodes. Many control nodes form a control plane. Control nodes are responsible for managing the cluster, while worker nodes are carrying the actual application payload. In case of an airplane the worker nodes could, for example, carry applications for controlling the cabin temperature, lights or the entertainment system. Each node is connected to each other via a reliable and highly available network plane (symbolized by a switch in the middle). All nodes consist of an operating system (Linux), a container runtime, a KubeProxy and a Kubelet. The operating system is very often a highly specialized Linux distribution, for instance Flatcar Linux[45] or Fedora's CoreOS[39]. Specialized Linux distributions have the huge advantage of being very minimal, high secure and low maintenancy requirements. They accomplish this via only shipping the Linux kernel, an init and service daemon like systemd[133] and a very small read-only base system. System upgrades are conducted through ostree image updates. The ostree is the operating system tree containing the base system. Contrary to other traditional Linux distributions like Debian, these specialized Linux container operating systems do not update software independently. Instead, the system upgrade is conducted via upgrading the whole base system at once. This procedure allows a very high testability, reliability and a lower maintenance compared to traditional Linux distributions. On top of the operating system is the container runtime, popular container runtimes are Containerd[25] or cri-o[30]. Docker was another famous container runtime for Kubernetes clusters, but it got recently deprecated in favor of Containerd[67]. Other components of every cluster are the KubeProxy and the Kubelet. The Kubeproxy is responsible for network connections between each node and establishes connections to each pod[65]. Kubelets are Kubernetes agents and are managing containers in each pod. Additional components of a control node are an etcd[36] instance, a Kube-Controller, a Kube-Scheduler and a KubeAPI instance. Etcd, is a highly available, reliable and distributed key-value store. Kubernetes uses etcd as its database for storing its actual and desired state. Every change on the Kubernetes cluster will be stored in the etcd cluster running distributed on all control nodes. Concensus is achieved via Raft, an advanced concensus algorithm proposed by the Stanford University[106][105]. Raft, that stands for replicant and fault-tolerant, implements consistency and partition tolerance as defined by Eric Brewer's CAP-theorem[8]. Eric Brewer's CAP theorem states that a distributed system can only implement two of three properties of the CAP theorem. These three properties are consistency, availability and partition tolerance. Consistency is when each node has the newest data, availability is when all read and write requests are always successful and partition tolerance means that the distributed cluster continues to operate even when not all nodes are available[5]. Raft guarantees consistency and partition tolerance, but it cannot guarantee availability due to unknown factors like network outages. Instead of using time, Raft uses election terms and roles for each node. Everytime an election is held, Raft will increment the term counter. On initialization all nodes have the role *follower* and the term counter on each node is zero. The first election term begins with one follower nominating itself as *candidate* and advertising this nomination to all other nodes. The other nodes will react with an acknowledgement and the candidate node will be promoted to a *leader*. The leader node sends heartbeats to every other node in the cluster. When a cluster node receives such a heartbeat the node's timeout will be reset. As long as the leader stays healthy nothing happens and the cluster works as usual. As soon as the leader dies the other nodes will stop receiving heartbeats. The missing heartbeart will lead to a timeout expiration. When this happens the expired node will increment the election term counter and nominate itself for a new leader and the process starts from new. This process ensures the partition tolerance as defined by Eric Brewer's CAP theorem. Data can be added to the key-value store only through the leader node and all followers will forward write requests to the leader node. The message is considered committed as soon as the majority of nodes acknowledged the write operation. Read operations can be done on every node. This is equal to Eric Brewer's consistency property of the CAP theorem. Further control node components are the Kube-Controller, the Kube-Scheduler and the KubeAPI. The KubeAPI provides an Application Programming Interface (API) endpoint for the cluster and external clients, for instance an admin who triggers changes on the cluster. If a pod gets created the Kube-Scheduler will schedule the pod on a free node. Kubernetes supports different scheduling and rollout strategies for deployments. Finally, the Kube-Controller implements different processes for controlling the cluster. For example, the Kube-Controller populates endpoints, takes care of service accounts, creates jobs or notices a node failure and responds accordingly[65].

3.5.3 Application Programming Interface (API) Overview

One of Kubernetes' biggest success factors is its extensibility. Kubernetes provides a good maintained Application Programming Interface (API) and implements necessary interfaces for extending Kubernetes with functionality. In this section, this thesis will highlight the most common API resources in the Kubernetes API, their purpose and their capabilities. For exploring the Kubernetes API this thesis uses a single node cluster bootstrapped via Minikube[89]. Minikube, written in Go, is a wrapper for bootstrapping a Kubernetes Cluster inside of one or more virtual machines. It allows a fast setup for local development and testing purposes, ideally for this thesis. Installing Minikube is rather easy: One just have has to download Minikube from its official website and run the following command: *minikube start –cpus 4 –memory 4096 –vm*. For running this command virtualbox is required as hypervisor. The command will download a slim virtual machine, provision it with 4 CPU cores and 4096 megabytes of RAM. The *–vm* flag will choose the virtual machine backend type. Minikube is able to provision a Kubernetes cluster directly on Docker, but this does not work on every operating system and is therefore Linux only. Running the start command of Minikube will setup a single-node cluster with its own control plane. The node will work as control node and worker node at the same time. Listing 3.6 shows the bootstrapping process of a single node Kubernetes cluster via Minikube. With the running cluster it is possible to automatically download and install the Kubernetes client tool *kubectl* and list all Kubernetes API resources via the command: *minikubectl kubectl api-resources – -o name*. Listing 3.7 lists the most important Kubernetes API resources. Pods consist of one or more containers. Configmaps store configuration data. Limitranges allow configuring resource constraints for deployments. Namespaces group resources and allow isolation between these groups. Nodes represent the node configuration. Storageclasses are classes for persistent storage, for example SSD hard disk storage or a network storage. Persistentvolumes hold information about persistent storage volumes for a given storage class. Persistentvolumeclaims represent the relationship between a container and a persistent volume. Podtemplates are re-usable templates for pods. Resourcequotas define a hard limit for resource consumption for a namespace. Secrets provide secret information, for instance secret configuration data. Services expose pods to the cluster-internal or external network. Daemonsets deploy pods on every cluster worker node. Deployments are a generic form of deploying pods, heavily modifiable. Replicasets are a mirror of the current state of replicas of a pod in the cluster. Statefulsets ensure that pods are deployed in a specific order and with a fixed set of volumes. Horizontalpodautoscalers provide the ability to up- or downscale pods dynamically depending on resource consumption. Cronjobs provide the ability to run pods on a fixed datetime. Jobs provide the ability to run pods just once. Ingresses are entrypoints for internal or external services, typically via HTTP/HTTPS. Networkpolicies provide isolation between pods on network layer. Poddisruptionbudgets manage budgets for pod failure or planned pod movements between nodes. Serviceaccounts manage access to API resources via a Role-Based Access Control (RBAC) model. Clusterrolebindings are bindings for roles on cluster level and service accounts. Clusterroles are roles on cluster level specifying the permissions for a service account via verbs: read, write, etc. Rolebindings are bindings for roles on namespace level, establishing the relationship between a role and a service account. Roles work on namespace level and provide permissions for a service account via verbs. The following sections will explain some of the prior mentioned API resources in more detail.

Listing 3.6: Running the minikube start command

```
$ minikube start --cpus 4 --memory 4096 --vm
minikube v1.24.0 on Arch rolling
> KUBECONFIG=/home/chris/.kube/config:
Automatically selected the virtualbox driver
Downloading VM boot image
> minikube-v1.24.0.iso.sha256: 65 B / 65 B [----------------]
    100.00% ? p/s 0s
> minikube-v1.24.0.iso: 225.58 MiB / 225.58 MiB  100.00% 25.23
    MiB p/s 9.1s
Starting control plane node minikube in cluster minikube
Creating virtualbox VM (CPUs=4, Memory=4096MB, Disk=20000MB)
Preparing Kubernetes v1.22.3 on Docker 20.10.8
> Generating certificates and keys
> Booting up control plane
> Configuring RBAC rules
> Using image gcr.io/k8s-minikube/storage-provisioner:v5
Verifying Kubernetes components
Enabled addons: default-storageclass, storage-provisioner
Done! kubectl is now configured to use "minikube" cluster and
    "default" namespace by default
```

Listing 3.7: Listing api resources in a Kubernetes Cluster (shortened)

```
$ minikube kubectl api-resources -- -o name
configmaps
limitranges
namespaces
nodes
persistentvolumeclaims
persistentvolumes
pods
podtemplates
resourcequotas
secrets
serviceaccounts
services
daemonsets.apps
deployments.apps
replicasets.apps
statefulsets.apps
horizontalpodautoscalers.autoscaling
cronjobs.batch
jobs.batch
ingresses.networking.k8s.io
networkpolicies.networking.k8s.io
poddisruptionbudgets.policy
clusterrolebindings.rbac.authorization.k8s.io
clusterroles.rbac.authorization.k8s.io
rolebindings.rbac.authorization.k8s.io
roles.rbac.authorization.k8s.io
storageclasses.storage.k8s.io
```

3.5.4 Pods

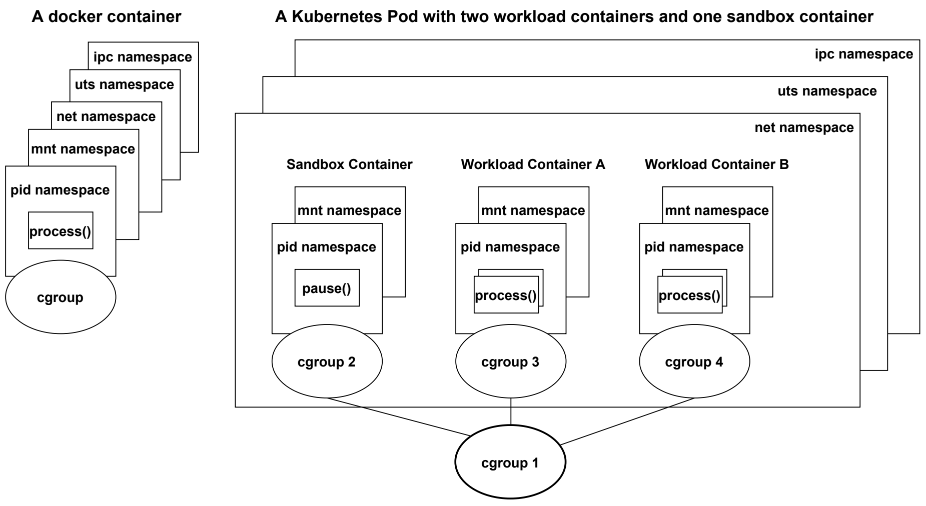


Figure 3.3: Comparing a Docker container to a pod[145]

Pods are the smallest manageable unit in a Kubernetes cluster and consist of one or more containers. Containers heavily rely on Linux kernel features such like namespaces, control groups (cgroups), kernel capabilities or the Extended Berkeley Packet Filter (eBPF). Figure 3.3 compares a Docker container with a Kubernetes pod and highlights the different namespace arrangements in each unit. The Docker container visualization on the left shows a Docker container with the kernel namespaces, its cgroup and the process running in the container. On the right the figure depicts a Kubernetes pod consisting of one sandbox container and two workload containers with one or more processes in each of them. The sandbox container works as initializer. It setups the net, uts and ipc namespaces as environment for all containers in the pod and creates an idle process. The pause process in the sandbox container does nothing[64] and the container acts only as initialization point for the pod itself. Environments do not need to be kernel namespaces. In theory, a sandbox Container Runtime Interface (CRI) operation could also start a virtual machine and setup a virtual machine environment[57]. The CRI allows full abstraction. For simplicity reasons, this thesis continues with containers only. The namespaces net, uts and ipc are responsible for setting up a network for all containers within the Kubernetes pod (net namespace), setting a host and domain name for the pod (uts namespace) and establishing Inter-Process Communication (IPC) between the containers. Sharing IPC capabilities between containers will be interesting in future chapters for avionics and reliability constraints. Workload containers implement the actual application payload. Figure 3.3 shows two workload containers with one or more processes in each of them. To mimic the same setup with Docker, there would be two containers in Docker with one or more processes in each of them.

Each container in the Kubernetes pod has its own mnt and pid namespaces for isolated mount tables and process isolation. This means that each container can mount its own additional external volumes and only sees the processes in the container, processes in other containers in the same pod or even on the host system are hidden for the processes in the container. Another major difference between the Docker container and the Kubernetes pod is the cgroups layout. The Docker container has just one cgroup. The cgroup is responsible for resource limiting, prioritization, accounting and controlling. Resources can be anything from CPU or RAM usage or even file system cache[125]. Prioritization means that processes in the container can be prioritized. Accounting refers to consumption logging and controlling means to trigger actions if certain thresholds get exceeded, for instance killing a process that consumes too much RAM. A Kubernetes pod has one cgroup per container and one root cgroup for the whole Kubernetes pod. This root cgroup makes it possible to control all container cgroups via one pod cgroup and to introduce outer limits for pod itself. Deploying a new static Kubernetes pod to a Kubernetes cluster requires a running Kubernetes (for instance via Minikube) and the Kubernetes client software *kubectl*. Pod API resources can be described either in JavaScript Object Notation (JSON) or Yet Another Markup Language (YAML). JSON and YAML are both data interchange formats. These data formats are passed to the Kubernetes API via Kubernetes' Representational State Transfer (REST) API. Representational State Transfer (REST) allows to contact the Kubernetes API over Hypertext Transfer Protocol Secure (HTTPS). Access to the Kubernetes REST API is protected via *X.509 certificates* and API objects can be added, modified or deleted via basic Hypertext Transfer Protocol (HTTP) methods, such like *GET*, *PUT* and *DELETE*. Listing 3.8 presents a YAML deployment file for a Kubernetes pod. Line 1-7 specify the API version, the API resource kind (pod) and additional metadata, like the name, target namespace and additional labels for easier selection and identification of the running pod. With line 8 the actual Kubernetes pod definition starts with a list of containers (line 9). The showed pod definition has only one container called *web* with the Docker image *docker.io/nginx:1.12.5*. Notice how the sandbox container is not listed in the pod definition. Sandbox containers are added to the pod automatically. Line 12 exposes the nginx webserver in the container via port 80 and Transmission Control Protocol (TCP) (a stateful network protocol for sending and receiving data). Line 16 specifies resource limits and requests for the container. The container can use up to 100 milli CPU (one CPU consists of 1000 milli CPU units) and 100 megabytes of RAM. Milli CPU allows to partition CPU consumption in reserved units. The limit limits the container CPU usage and the requests field tells the Kube-Scheduler how much the container needs. If the limits and request matches each other, the Kubernetes pod is sorted into the Quality of Service (QoS) class *Guaranteed*[71]. QoS classes are used for scheduling containers on the cluster. Kubernetes supports the following QoS classes: *Guaranteed* states that the container will have a guaranteed resource consumption. If a cluster node is overprovisioned, other less important containers will be re-scheduled on a different node or stopped in favor of guaranteed containers. *Burstable* states that a container is scheduled on a cluster with a fixed request value, but the container is allowed to create resource consumption bursts. This is, for instance, the case when a container, suddenly, gets a very high load due to higher usage. An example in the aviation domain could be a container responsible for flight entertainment in the passenger cabin. The flight entertainment may have times of higher and lower usage depending on the flight length (sleeping passengers etc.). A container is sorted into the *Burstable* class, when the container has at least one requests value for memory or CPU. The *BestEffort* QoS class is used for all containers without a limit or request value, these containers are scheduled on the cluster node per best effort and may be stopped or re-scheduled on a different node every time. If a container uses more than the specified RAM the cgroup will notice this and kill the process in the container. This behavior is different to reaching the CPU limit. When the CPU limit is reached the container will be throttled. Throttling a CPU is, when a process gets less CPU time from the operating system. Many processes are not being executed in parallel by a single CPU core, instead the CPU schedules execution time for each process with different queueing methods. Listing 3.8 shows only very small set of configuration options. The full Kubernetes pod API consists of more than 1000 possible configuration options reaching from liveness and readiness probes, to security modifications, mounting volumes and devices from the host system, more advanced network configuration and many other possible configurations that are useful for running containers in a big scale in production (retrieved via running the command: *minikube kubectl explain –-recursive | wc -l*).

Listing 3.8: An example pod definition in YAML

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  namespace: default
  labels:
    app: example-pod-application
spec:
  containers:
  - name: web
    image: docker.io/nginx:1.21.5
    ports:
    - name: web
      containerPort: 80
      protocol: TCP
    resources:
      limits:
        cpu: 100m
        memory: 100M
      requests:
        cpu: 100m
        memory: 100M
```

3.5.5 Namespaces, LimitRanges and ResourceQuotas

A namespace groups resources logically for better filtering, selecting and operating on Kubernetes API resources. Most Kubernetes API resources can be namespaced (placed into a namespace). On their own, namespaces provide no network isolation. For strict network isolation a Container Network Interface (CNI) plugin is necessary and the definition of network policies via the networkPolicy API resource. Listing 3.9 depicts a definition of one namespace called entertainment, a limitRange and resourceQuota. The namespace works as environment for other API resources, like deployments or pods. The limitRange API resource allows to set limits and requests for each created container in the selected namespace. ResourceQuotas are responsible for setting hard limits for the namespace as a whole. The resourceQuota in Listing 3.9 specifies a maximum limit of 10 pods in the entertainment namespace. Furthermore, it sets a limit and request value for the CPU and RAM for the entertainment namespace to one CPU and one gigabyte. Thus, the Kubernetes-Scheduler enforces these limits, making it impossible to schedule more than 10 pods in the entertainment namespace and constantly monitores the resource consumption. If a pod exceeds its limits, the pod will be either throttled (in case of CPU bursts) or the process in the pod container will be killed in case of too high

RAM consumption. These three API resources enforce a very tight control over resources and enhances the reliability of applications running in the cluster.

Listing 3.9: Namespace definition for a namespace called entertainment with limitRange and ResourceQuota

```
apiVersion: v1
kind: Namespace
metadata:
  name: entertainment
---
apiVersion: v1
kind: LimitRange
metadata:
  name: entertainment-limitrange
  namespace: entertainment
spec:
  limits:
  - default:
      cpu: 100m
      memory: 100M
    defaultRequest:
      cpu: 100m
      memory: 100M
    type: Container
---
apiVersion: v1
kind: ResourceQuota
metadata:
  name: entertainment-rq
  namespace: entertainment
spec:
  hard:
    pods: "10"
    requests.cpu: "1"
    requests.memory: 1G
    limits.cpu: "1"
    limits.memory: 1G
```

3.5.6 Deployments, Statefulsets and Daemonsets

Multiple instances of the same Kubernetes pod are called a replicaset. A deployment manages pods and replicasets. With deployments it is possible to define a desired state of an application. Kubernetes tries to match this desired state via spinning up the pods and replicas of these pods, as defined in the deployment's definition. Listing 3.10 demonstrates a Kubernetes deployment representation in YAML. Line 18 to 31 are similar to the pod definition in Listing 3.8. The most important parts of the deployment definition are from Line 9 to Line 16. These lines specify the replica count, a label selector and the begin of the pod template definition. The replica count in line 9 sets the number of replicas. This means that Kubernetes tries to schedule three pods of a replicaset, running the same containers. The label selector in line 10 selects the pod definitions that should be scheduled by label. Line 16 shows the matched labels. Statefulsets and daemonsets are very similar to deployments. The major differences between deployments and statefulsets ensure that all pods are scheduled in a persistent order and pods get a number suffix instead of a random string suffix when scheduled on the cluster[73]. A persistent order and pod uniqueness is notably important for stateful applications with storage or sticky network sessions. A daemonset schedules the pod on every node. This behavior is especially useful for logging or storage management daemons running on every node[66].

Listing 3.10: An example deployment with three replicas

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
  namespace: default
  labels:
    app: example-pod-application
spec:
  replicas: 3
  selector:
    matchLabels:
      app: example-pod-application
  template:
    metadata:
      labels:
        app: example-pod-application
    spec:
      containers:
      - name: web
        image: docker.io/nginx:1.21.5
        ports:
        - name: web
          containerPort: 80
          protocol: TCP
        resources:
          limits:
            cpu: 100m
            memory: 100M
          requests:
            cpu: 100m
            memory: 100M
```

3.5.7 Configmaps and Secrets

Applications require configuration. Either the configuration has been embedded into the application or the configuration happens via an external source, for instance a configuration file or a configuration service that offers a configuration via network. Kubernetes does support both methods. With OCI containers it is possible to embed a default configuration directly in the container image. However, this approach is very unflexible and static, because it does not allow dynamic reconfiguration. Dynamic reconfiguration can be achieved via setting new configuration via the network or via swapping the configuration files. The first option requires more development work, because a dynamic reconfiguration over, for example a web interface, has to be embedded into the application. The second option only requires to read a file. This file read can be either triggered through a file change (file watcher) or via an application restart. It also means, that the container runtime must implement methods to swap configuration files. One way to solve this in Docker, is via attaching volumes and placing files on the host system. These files are then mounted into the running container. Another option is to inject environment variables into the container. Kubernetes is not so different to Docker and solves the problem with the same methods. Nonetheless, Kubernetes adds a few security features that are missing in Docker. Similar to Docker, Kubernetes offers multiple ways to inject configuration data into containers. The first common approach is injecting the data via environment variables directly in the pod or deployment configuration. These environment variables can either be loaded from the API resource definition itself or dynamically from another API resource, such like configmaps or secrets. Configmaps and secrets store container configuration in the Kubernetes cluster itself. To be more precisely, the configuration data is being stored in the highly available key-value store called etcd. The main difference between configmaps and secrets lies in the way how Kubernetes stores the data. Configmaps are stored in plaintext and secrets are stored as base64. Data encoding via the base64 algorithm have been introduced in RFC4647[60]. A Request For Comments (RFC) is a publication in a series of technlogical documents from the Internet Engineering Task Force (IETF)[115]. Base64 is a data encoding algorithm for submitting data in a compact form. It does not implement encryption and base64 encoded data can always be decoded. Kubernetes uses base64 as placeholder algorithm, suggesting that platform administrators have to provide their own secret management service for better security. Nevertheless, Kubernetes secrets are better protected by the Kubernetes API than configmaps and adding or reading secrets require more permissions in Kubernetes' API. Listing 3.11 is a copy from Listing 3.8 with one little change. Listing 3.11 introduces a set of two configuration variables to the pod via importing them into the container's environment through a configmap.

Listing 3.11: Embedding configuration data via a configmap in a pod

```
apiVersion: v1
kind: Pod
metadata:
    name: example-pod
    namespace: default
    labels:
      app: example-pod-application
spec:
    containers:
    - name: web
    image: docker.io/nginx:1.21.5
    envFrom:
      - configMapRef:
           name: example-configmap
    ports:
    - name: web
      containerPort: 80
      protocol: TCP
    resources:
        limits:
        cpu: 100m
        memory: 100M
        requests:
        cpu: 100m
        memory: 100M
---
apiVersion: v1
data:
  config.yaml: |
    NGINX_HOST: example.com
    NGINX_PORT: 80
kind: ConfigMap
metadata:
  name: example-config
  namespace: default
```

3.5.8 Services, Ingresses and NetworkPolicies

Network plays a huge role in a distributed container orchestrator. Hence, it is not a surprise that Kubernetes offers advanced network functionality. For this, Kubernetes has several API resource types. In this subection, this thesis will give a short overview over the most used network related Kubernetes API resources: Services, Ingresses and NetworkPolicies. Services provide an abstract way to expose deployed containers to the internal cluster network or any external network. A service can be one of four service types: clusterIP, nodePort, loadBalancer and externalName. ClusterIP services expose the deployments to the cluster-internal network[72] via acquiring an internal cluster IP for the service and routing all traffic through the KubeProxy to the pods behind the service. The nodePort service type exposes the deployment behind the service to external networks via one of 65535 ports on the Linux node. A deployment can then be reached via the node's IP address and the opened node port. LoadBalancer services are able to use loadbalancers from cloud providers (if Kubernetes is running in the cloud) or from deployed load balancing services in the cluster. Projects like KubeVip[75] or Metallb[88] provide this functionality via constructing Address Resolution Protocol (ARP) or Border Gateway Protocol (BGP) based routing on-premise. Address Resolution Protocol (ARP) helps with discovering the hardware address (Media Access Control (MAC) address) for an IPv4 address and has been defined in RFC826[109]. Metallb implements ARP via deploying, so called speaker pods, to every node. These speakers announce a load balancer IP address through an ARP announcement. A gateway catches these ARP announcements and routes the traffic to the announced IP address via their Media Access Control (MAC) address. This behavior of Metallb, is also called layer 2 mode. If the leader node dies, the floating IP address will be moved to a different node[87]. In Border Gateway Protocol (BGP) mode the Metallb speakers will establish BGP peering sessions between each cluster node and the network router. These peering sessions are then used to advertise the IP addresses[86] The KubeProxies are responsible for load balancing the incoming traffic to all pods behind the Kubernetes service. The last Kubernetes service type is the externalName. ExternalName services are enabling Kubernetes services to forward to cluster-external Fully Qualified Domain Names (FQDNs). The next important Kubernetes API resource is the ingress. An ingress is comparable to a service. The ingress handles incoming traffic and forwards it to a service backend. During this process the ingress is able to act like a middleware. A middleware allows filtering, restricting or modifying the incoming traffic. Most common usecases for ingresses are adding basic authorization for HTTPS endpoints or playing the HTTPS endpoint for a HTTP backend. In the field of aviation an ingress is not that important. NetworkPolicies are important. NetworkPolicies allow isolating namespaces or pods from each other on a network level. Isolation is important for security and reliability reasons. For instance, a flight entertainment system should be isolated from a system to control the lights in the passenger cabin. In case of a security breach of the entertainment system, this would prevent the attacker from gaining access to other systems. Or in case of a misbehaving system, without an actual attacker, the isolated namespace would prevent the system from doing harm to other services on the cluster, for example bombarding them with network traffic and leading to performance regressions. Listing 3.12 presents a Kubernetes networkPolicy for rejecting all incoming and outgoing traffic within the namespace default, except for DNS traffic. The networkPolicy matches all namespaces, but selects only pods with the label value kube-dns and the port 53 with protocol User Datagram Protocol (UDP) in use.

Listing 3.12: Example networkPolicy that rejects all traffic in a namespace except for DNS requests

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: reject-all-traffic-except-dns
  namespace: default
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
  ingress: []
  egress:
  - to:
    - namespaceSelector: {}
      podSelector:
        matchLabels:
          k8s-app: kube-dns
      ports:
      - port: 53
        protocol: UDP
```

3.6 Secure Software Supply Chains

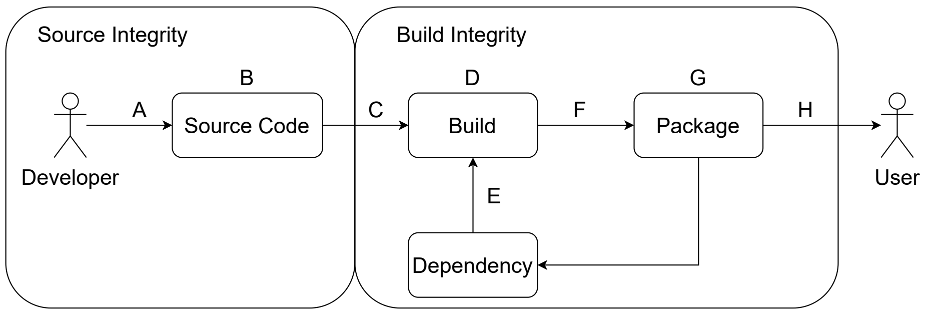


Figure 3.4: Supply chain overview with all possible attack points (A-H)[58]

Over the last 10 years software supply chain incidents had such an impact on the IT industry, that US president Joe Biden proclaimed an executive order for software supply chains security[37]. Incidents, like the SolarWinds software supply chain infiltration, had a direct impact on over 18,000 customers world wide and multiple US government agencies. Nowadays, software is everywhere and although planes are not directly connected to the internet, their software supply chains are vulnerable. Software and Hardware supply chains are not very different. While a hardware supply chain consists of all steps from harvesting the necessary resources, to production, assembly, and distribution, a software supply chain consists of one or more developers writing source code, embedding other software as dependencies, testing, pushing, packaging, and distributing it. Each of these steps can reveal a possible weak spot for attackers. Especially, the wide direct or indirect use of open source software, can be a challenge for companies. Recently, the National Aeronautics and Space Administration (NASA) had to announce, that their Mars helicopter *Ingenuity* is not vulnerable to a security vulnerability in an open source Java logging library[93]. It is not uncommon for aviation companies to either use open source software during their development process or even in flight systems. For instance, in 2007 Airbus announced an open source toolkit for mission critical applications[46]. Figure 3.4 visualizes each step of a supply chain and all vulnerable spots (A-H). The software supply chain starts with one or more developers writing the source code. Modern software development involves pushing code differences (commits) to a Source Code Management (SCM) system (A in Figure 3.4). Possible vulnerabilities involve an attacker in the company's network with access to the SCM system (for example Git, SVN or CVS), a stolen laptop with the developer's SCM credentials or code from a malicious code contributor (more common in open source software). In May 2021 researchers of the University of Minnoesota managed to contribute vulnerable code to the Linux kernel SCM[35][58]. B in Figure 3.4 involves a direct access to the SCM system. Contrary to A, where access to the SCM is achieved through a developer, B involves exploiting the SCM system directly and altering the content of the code in the SCM. Another famous example for such an incident is the corruption of the PHP development team's SCM system in 2021[17][58]. PHP is one of the core languages behind the World Wide Web (WWW) and had a market share of 79.2%[140]. There is no imagination needed for what could have happened, if this malicious code would have been distributed to all PHP systems world wide via a new version release. The same can happen with any other programming language the aviation industry relies on (C, Ada, C++[130]). During the next step in the software supply chain the code gets build, either via the developer or, the more common approach, via a build system. In this step, multiple attacks can happen. First of all, the code that should be build could get replaced during the build process. One occassion of such a scenario was, when Webmin's build system got feeded with malicious code, thus building a vulnerable Webmin release (C in Figure 3.4)[151][58]. Webmin is a web-based interface for server administration, hence it is very critical infrastructure, because it allows direct access on the server with admin privileges. Some other scenario is a direct corruption of the build system, accepting correct code, but altering it directly on the build server or injecting additional malware into the normally harmless build. This scenario happened in the big SolarWinds hack mentioned earlier[131][58]. Vulnerability E in Figure 3.4 describes the addition of malicious software dependencies. The project's software supply chain is secure, but software dependencies have been added, that are either poorly maintained or that got victim of a software supply chain attack themselves. Some attackers, inject vulnerable dependencies via contributing to the software via a harmless patch with a new dependency and then after a few weeks they add a vulnerability to the dependency and they hope that the victim will update their dependencies without double-checking the updated dependencies. Event-Stream has been a victim of this kind of attack[120][58]. F to H in Figure 3.4 refer to the distribution stage of a software supply chain. In this step, the compiled software gets distributed to users or deployed to a target system. In April 2021 the company Codecov announced that their Google cloud storage account got breached into via stolen credentials[110][58]. Attackers used the stolen credentials to upload malicious files to the software distribution system (F in Figure 3.4). G in Figure 3.4 refers to a compromised package repository. Researchers of the university of Arizona successfully attacked package managers via hosting malicious package mirrors[15][58]. The last possible weak point is the user. An attacker is able to exploit that a human user or developer might make mistakes. One of these mistakes can be spelling a software name wrong. Attackers can use this knowledge and upload malicious software with similar written names and hope that a developer makes a mistake while adding dependencies or the user makes a mistake while downloading the software. The name of this method is called typosquatting and a good example for this is, is one incident with the widely used Javascript package manager npm[124][58]. All these incidents lead to one final question: How to protect software supply chains? Due to the increased appearance of software supply chain compromises and the increasing pressure of the US government the Linux foundation came up with a new standard called Supply chain Levels for Software Artifacts (SLSA)[127]. SLSA proposes four different levels for measuring and classifying software supply chain security. These levels are considered as milestones leading to full software supply chain protection with SLSA level four. The first SLSA level states that the software must be build scripted and the supply chain must attach provenance. Provenance is metadata about dependencies, the software itself and the build process[126]. Scripted means that the supply chain must be automated. Automation protects against human errors and is a foundation for all other SLSA levels. Attaching provenance or a Software Bill of Materials (SBOM) has several advantages. First, it allows to store metadata about the dependencies and the software itself. With the help of this additional metadata it is possible to search for security vulnerabilities in the product via version numbers and other software identifiers[108]. Second, it gives a complete list of contents. This allows identifying missing content and unwanted content (for instance malicious dependencies that got injected through a hacked build process). Third, it gives an overview over all used licenses. With the wide usage of open source software it is necessary to gain information about the software licenses of dependencies to avoid license infringements (for example using an open source license that prohibits the use in proprietary software)[147]. A Software Bill of Materials (SBOM) must be machine readable for automation and must evaluate the gained data automatically. Level two of SLSA requires the use of SCM and a hosted build process with authenticated provenance[126]. Through SCM it is possible to track every step of work on the source code in a commit log. Some SCM like git even allow cryptographically signing each commit and thus authenticating every commit in the SCM. Signing commits offers additional security for verifying that the code has been submitted by a list of allowed developers. Having the build process hosted, secures the build process and isolates it to a single host. A single host is a lot easier to protect and harden than a higher number of developer workstations or laptops. The last requirement for SLSA level two is authenticated provenance. Authenticated provenance means that the provenance has to be signed cryptographically. This enables other systems to validate the provenance's integrity and origin. SLSA level three dictates further enhancement to the security of the build process and build host. Additional enhancements of the build environment suggest an isolation between build processes and the use of an ephemeral environment. The first ensures that a build process cannot be interfered with via another build process. The second protects the build process from accidentally using persistent artifacts from a past build process. Moreover, level three recommends to have the build pipeline configuration in source control[128]. Storing the build pipeline in source control via SCM implements a change log for it. This allows to understand changes. Lastly, SLSA level four has the most requirements and is most difficult to achieve from all levels. Artifacts in SLSA level four must be reproducible, hence the software must compile deterministically to the same executable[128]. This can be tested with calculating the cryptographical checksum of two separately compiled binaries (Being reproducible comes very handy in aviation, because DO-178B requires deterministic software and a comprehensive test suite). Additionally, builds have to be parameterless and hermetic. Parameterless means that the build process must be only affected by the build script in the source control and by nothing else. Being hermetic means that the build environment has no network access and that the full build process has been declared up front via immutable references[128]. Other requirements for level four can be find in the security domain: All changes should be two person reviewed, the system must underwent continuous vulnerability and security checks, physical and network access must be protected and logged and only super users should be able to modify any of these security constraints[128].

Chapter 4

Cloud-Native and Aviation

4.1 Related Projects

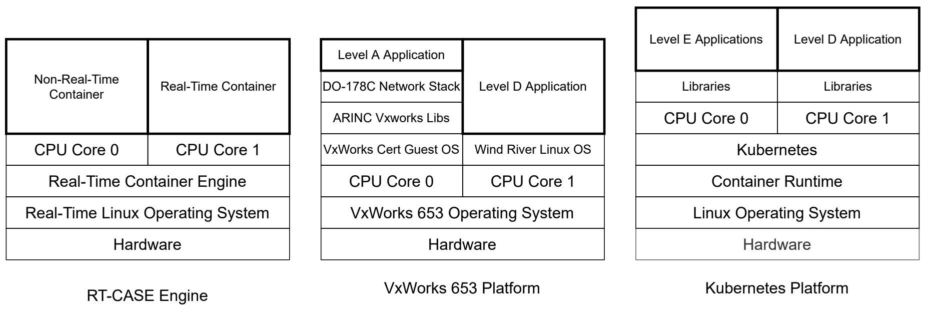


Figure 4.1: Direct Comparison of the Real-Time Case Engine by Marcello Cinque and Gianmaria De Tommasi (University of Naples, Italy) with Wind River's VxWorks 653 platform and the proposed Kubernetes Platform[20][153][148]

This chapter fulfils the purpose of connecting the presented aviation system designs, challenges and opportunities from chapter 2 with the first step into the Cloud-Native world in chapter 3. The big question of this part of the thesis is how a future aviation system design could look like and how emerging technologies from the Cloud-Native iniative could be beneficial for aviation. Software development in aviation is expensive, complex, must follow certain requirements and is splitted into two domains: Airborn software and ground-based software. Airborne software certification guidance is defined in DO-178C (the successor of DO-178B). Ground-based software is defined in DO-278A. These two certification guidances are being supported by a various number of additional documents, such like:

DO-330 Software Tool Qualification Considerations

DO-331 Model-Based Development and Verification

DO-332 Object-Oriented Technology and Related Techniques

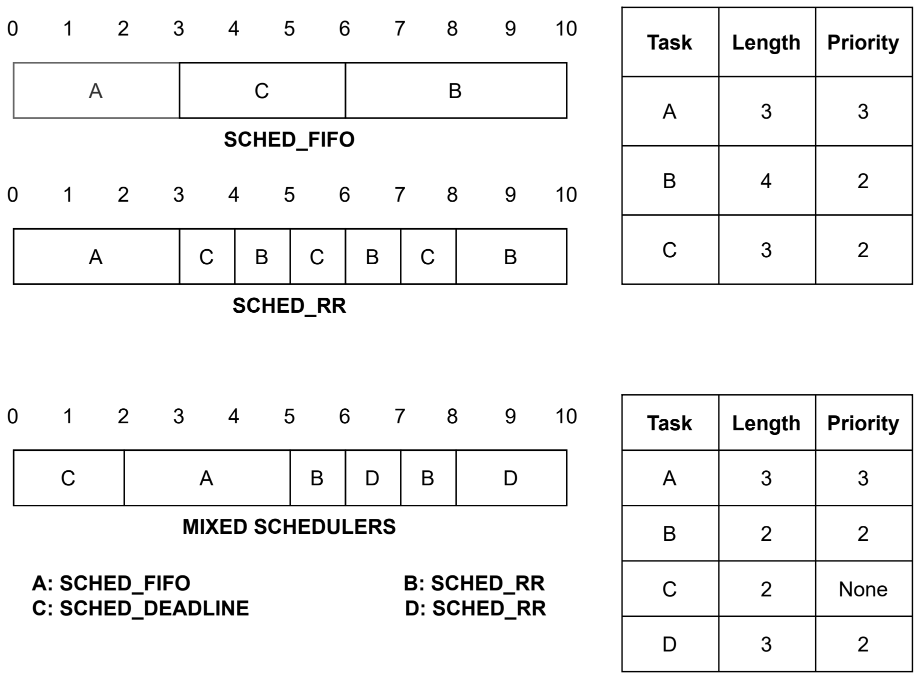
DO-333 Formal Methods

According to DO-178C and DO-278A software is categorized in different Design Assurance Level (DAL), also known as Item Development Assurance Level (IDAL) (a complete list can be found in chapter 2). A is the most important software level, critical to human life, and software level E is the least important software level with no harmful incidents. DO-178C specifies for each DAL tracing requirements. Tracing refers to the strict requirement of traceable code. Every code snippet must be traceable to a required feature and test case[10]. One famous example for an existing feature in documentation, but missing feature in the code base, is the Ariane 5 disaster in 1996[31]. Modern software development paradigms like Test Driven Development (TDD) can encourage software developers to increase the test coverage and implement only necessary features[107]. Another important requirement in DO-178C is the presence of deterministic build and execution behavior. Software builds and execution must be deterministic. Undeterministic behavior could lead to catastrophic events in aviation. One modern key concept supporting this requirement is binary reproducibility. Binary reproducibility states that there must be correspondence between the application and the source code[113]. This means that the build process must be deterministic and must produce the same binary bit by bit. Avation may profit from reproducible builds for their DO-178C and DO-278A certification processes. Reproducible builds is an important factor in the SLSA specification for secure supply chains as well. Containers help with achieving reproducible builds via providing an isolated and ephemeral build environment. Interestingly, Cloud-Native containers are not that popular in the aviation industry. This is astonishing, because containers have, like previous chapters have showed, many potential features that can help with many challenges from the aviation industry. The analysis of the different aviation system designs clearly shows a demand for cost reduction, lower development to production time, and Total Cost of Ownership (TCO) reductions. These challenges are very similar to the challenges from the cloud computing industry with the slight difference that aviation software must satisfy software reliability constraints as defined in DO-178C. Existing solutions like the Wind River VxWorks 653 platform provide an answer for some of the prior stated challenges. VxWorks 653 runs already on more than 90 aircrafts, including the Airbus A400M or the Boeing 787 Dreamliner[153]. The platform is DO-178C compliant, supports an unmodified Linux guest operating system, real-time capabilities, a multi-core scheduler with hardware virtualization assistance and a unique IMA design layout[148]. IMA design is being achieved via running a central proprietary module operating system in the flight cabinet. This operating system supports Aeronautical Radio Incorporated (ARINC) ports, ARINC health management and support for ARINC configuration via Extensible Markup Language (XML). Each physical CPU core manages one dedicated virtual machine with different operating system workloads, depending on the application's DO-178C compatible software level. This allows running multiple applications in parallel on different CPU cores on the same physical host system with a maximum of safety and reliability, but it also leads to a possible target for improvement. Running virtual machines is very costly and inefficient. Multiple studies have proven that virtual machines are significantly less performant than containers or bare-metal systems[122][40]. Other papers even actively suggest using real-time containers for large-scale mixed-criticality systems[20]. A mixed-criticality design comes very close to the VxWorks 653 platform as the VxWorks 653 platform is capable to run different workloads in different virtual machines with different certifications for different DO-178C software levels[153]. Figure 4.1 presents a direct comparison between Wind River's VxWorks 653 platform (middle), the proposed Kubernetes platform of this thesis (right) and previous work from Marcello Cinque and Gianmaria De Tommasi (university of Naples, Italy) with their proposed real-time Linux container engine RT-CASE (left). Mixed-criticality systems have in common that they allow running multiple workloads with different criticality levels on the same hardware. The main goal of these systems is to reduce costs, space and power consumption, while meeting stringent non functional requirements[20]. Wind River's VxWorks 653 platform manages this balancing act with a traditional IMA approach and a proprietary hypervisor. Application payload is deployed on operating system partitions, scheduled by the VxWorks module operating system, each partition only executes when their allocated time slice is active (standard ARINC time-preemptive scheduling)[152]. The VxWorks module operating system executes on kernel level and the partitions with different criticality payloads operate in user mode. Different scheduling algorithms are supported: The standard ARINC time-preemptive scheduling, a mode-based scheduling algorithm and a scheduling mode called ARINC plus priority-preemptive scheduling (APPS)[152]. In time-preemptive scheduling mode partitions execute until their time slice expires[152]. Time is prioritized and every process gets exactly the amount of time as it has been orchestrated. This behavior is different to APPS, where a priority-based scheduling mode is being used. APPS allocates a priority level to every process and processes with a higher priority are preferred. If a process with higher priority needs CPU time, the lower-prioritized process must wait. During mode-based scheduling the partition schedules can get configured statically and selectively enabled dynamically on-demand[152].

The left structure in Figure 4.1 shows a diagram of the RT-CASE container engine for running real-time containers in a mixed-criticality system. The RT-CASE engine is a prototype for a multi-criticality system based on a patched real-time Linux kernel with additional libraries for scheduling LXC Linux containers for different criticality levels[20]. All containers run in unprivileged cgroups on a patched real-time Linux kernel. The core component of the real-time container engine is the Real Time Application Interface (RTAI), an hardware abstraction layer for the Linux kernel for running real-time tasks, by the Department of Aerospace Sciences and Technologies Milan, Italy[116]. RTAI is under active development and consists mainly of two parts: A real-time patch for the Linux kernel with hardware abstraction layer and a "broad variety of services which make real-time programmers' lifes easier"[1]. An early implementation of the RT-CASE engine involves patching RTAI core libraries for providing container isolation for different priority levels[19]. This type of isolation for priority classes is necessary for running the containers via a preemptive fixed-priority First-In, First Out (FIFO) scheduler on Kernel level[19]. Isolation is being established via remapping each real-time task onto separated time intervals. Remapping allows the preemptive fixed-priority FIFO scheduler to run each task in a specific order (First In, First Out). Tasks with a higher priority level are preferred. A real-time task with high priority level can interrupt any task beneath them. This approach also allows priority inheritance, similar to virtual machines[19]. In 2019, Marcello Cinque conducted final tests with the RT-CASE engine. Different experiments were performed. In these experiments tasks with different priority levels were executed on machines with different CPU utilization. Each task were tested for overtime and overrun situations. If a task exceeds their Worst-Case Execution Time (WCET) this is known as overtime. Overrun refers to a missed deadline. The paper shows that three high criticality tasks met all overtime and overrun constraints and seven medium criticality tasks violated the overtime constraint (with one exception violating the overrun constraint, this has been a task with the lowest priority)[21]. This result definitely matches this thesis expectation that containers are a feasible solution for avionics. Interestingly, Marcello Cinque and Gianmaria De Dommasi mention in their article Kubernetes as solution for mapping task names, containers and configuration[20]. Kubernetes as possible solution is illustrated on the right of Figure 4.1. The concept of using Kubernetes as distributed container orchestration platform is not so different to the RT-CASE engine with real-time containers. Most research regarding real-time containers on Linux form a good entrypoint for further research, because implementing a runtime engine for real-time containers, choosing the right scheduler, and establishing isolation is not enough for a system, that must fit real world challenges. In reality, a system should be easy to use, easy to maintain, secure, and cost-effective if it aims to get attention outside of the academic world. Kubernetes has this attention already in the cloud computing industry. What is missing is the missing link and potential in other industries like the automation industry, in particular in fog- and edge-computing, or in the aviation industry. The combination of real-time containers and Kubernetes has been already topic of many researchers[47][129][42][61]. This thesis tries to focus on the orchestration and reliability aspects with Kubernetes and tries to discover potential benefits for the aviation industry in the Kubernetes API. In the previous chapters, this thesis has explored various resource types of the Kubernetes API. These resource types show a huge potential for aviation systems. One of the main questions is, if container systems can provide the same reliability safeguards than virtual machines. Reliability safeguards are implemented through the backbones of containerization in Linux: Namespaces and cgroups. Additionally, Kubernetes provides different QoS classes. Listing 3.10 depicts a deployment with three replicas and guaranteed CPU and RAM allocations. Replicas can be deployed on different nodes through anti affinity rules, matching one pod (set of containers) per worker node. Through a Kubernetes service resource it is possible to loadbalance the traffic over all three replicas. If a replica fails, Kubernetes's kube-scheduler automatically tries to restart the pod. Failing pods can be detected either via a process exiting with a return code unequal to zero, or via a failing readiness or liveness probe or via custom monitoring.

4.2 Real-time capabilities in Linux

The Linux operating system itself is not real-time capable, hence various projects emerged to provide the Linux kernel with necessary patches for enhanced real-time capabilities. The most prominent example for such a patch set is the Linux Foundation's *PREEMPT_RT* project. Projects like the RTAI project from the Department of Aerospace Sciences and Technologies in Milan are extending the existing *PREEMPT_RT* project with additional libraries or smaller changes in the real-time Linux kernel patch set. According to the Linux Foundation an real-time operating system is a system that is able to run one or more real-time tasks, before a certain deadline[135]. Other requirements for a real-time operating system are a fast execution speed, low latencies and most importantly deterministic behavior. These real-time tasks should be preemptive, this means that the task is running with a higher priority than any other task and should not be disrupted by tasks with a lower priority. The real-time Linux kernel achieves this with a set of features, that are missing in the mainline Linux kernel. The mainline Linux kernel comes with just 40 priority levels, or so called *nice* levels, and implements a system call to set the level for a process[134]. System calls are the fundamental interface to the Linux kernel[132]. The level with the highest priority has the value -20, the lowest priority is +19 and the default value is 0. *PREEMPT_RT* extends the *nice* levels with 100 real-time priority levels reaching from 0 to 99, while 0 is the lowest and 99 is the highest priority level. Real-time priority levels are preferred over the traditional *nice* levels and are protected via the system call *mlock*[4]. Mlock allows locking the used memory in the RAM, thus the process cannot swap memory on the hard disk (leading to a decreasing execution time)[141]. Critical sections are preemtable through special real-time mutexes, this does include preemptable interrupt handlers. An interrupt disrupts an executing process. This is bad in real-time systems, because of two reasons. First, a real-time task should be able to interrupt an interrupt and execute code in the necessary time and second a high priority real-time task should not get interrupted by a task with a lower priority. This makes process scheduling very important on a real-time operating system. Therefore, it is not a surprise that a real-time operating system comes with a new set of schedulers. A scheduler determines time slices for process for their execution. The *nice* system call is part of the scheduler and the mainline Linux kernel has a small set of non-real-time capable scheduling policies: *SCHED_OTHER*, *SCHED_IDLE* and *SCHED_BATCH*. On default the mainline kernel uses the Completely Fair Scheduler (CFS)[118]. The CFS aims to maximize CPU utilization through scheduling process fairly on the available processors, respecting the different scheduling policies. *SCHED_OTHER* is the default scheduler for all processes, *SCHED_BATCH* is designed for a higher throughput and *SCHED_IDLE* is a scheduling policy for processes with a lower priority than the weakest *nice* level[119]. These scheduler policies are not suitable for a real-time operating system, hence other scheduler policies have been introduced via the *PREEMPT_RT* patch set. Three scheduler policies are capable of real-time scheduling: *SCHED_FIFO*, *SCHED_RR* and *SCHED_-DEADLINE*. Figure 4.2 visualizes the different scheduling policies and their mechanisms. The first time slice shows what happens if all tasks on the right have been scheduled via the *SCHED_FIFO* policy. On the right of the first task slice is a table with the tasks A, B and C with different lengths and real-time priorities. During the *SCHED_FIFO* policy the tasks are scheduled with respect to their real-time priority level. Task A is scheduled at first, because task A has the highest real-time priority. After scheduling task A there are two tasks with the same real-time priority left. In this case, the task which comes first will be served first (in this case task C before task B). The second time slice shows the same scenario with all tasks running under the *SCHED_-RR* policy. *SCHED_RR* defines a fixed time slice for each task. Although the scheduler sets the fixed time slice for each task to one, task A is served first again, because task A has the highest priority over all over tasks. When task A has finished, the other tasks are being run according to the fixed time slice in a round-robin cycle. The last time slice in Figure 4.2 shows a mixed scheduler scenario. Linux allows setting different scheduler policies for different processes, hence it is absolutely common to run processes with different scheduling mechanisms. Running processes with different schedulers makes the real-time patched Linux kernel to a mixed-criticality system. On the right of the bottom time slice there are now four tasks. Task A is being scheduled with the *SCHED_-FIFO* policy, task B and task D run through the *SCHED_RR* policy and task C is managed by the *SCHED_DEADLINE* policy. Attentive readers will see that task C has no real-time priority. A real-time priority is in case of task C not necessary, because *SCHED_DEADLINE* is being implemented through the Global Earliest Deadline First (GEDF) scheduling algorithm. Contrary to the other scheduling policies which are fixed-priority based, the GEDF algorithm schedules tasks with the global earliest deadline first. Global refers to all processor cores in a system, the word clustered would refer to a subset, partitioned would mean that each scheduler manages a single CPU and arbitrary means any other CPU set[9]. The time slice on the right of the bottom table shows the schedule times for each task. Task C is run first, because task C is managed via the *SCHED_DEADLINE* policy. *SCHED_DEADLINE* can preempt every other task and runs the tasks in this priority class with their earliest deadline first. There is just one task in this priority class, thus task C runs first. When task C has finished, task A is being scheduled, because task A runs through the fixed-priority based *SCHED_FIFO* policy and has a higher priority class than the tasks B and D. The tasks B and C run in a round-robin cycle, because they share the same scheduling policy and the same real-time priority.



Task	Length	Priority
A	3	3
B	4	2
C	3	2

Task	Length	Priority
A	3	3
B	2	2
C	2	None
D	3	2

Figure 4.2: Different real-time scenarios with different real-time schedulers

Out of all real-time schedulers, the *SCHED_ DEADLINE* policy is the preferred one nowadays, because fixed-priority schedulers are affected by the priority inversion problem[134]. The priority inversion problem occurs, when a low-priority task is blocking a resource, for instance via a semaphore or a mutex. If a high-priority task tries to access the resource, the high-priority task must wait due to the semaphore or mutex. The situation gets more problematic when a medium-priority task with access to a different resource gets scheduled, because now the medium-priority task has a higher priority than the low-priority task, hence leading to an execution of the medium-priority task. The high priority task will not get scheduled at all. There are two different solutions for this problem. The first solution is called priority ceiling. Priority ceiling protects a resource with a priority-gate. This means that any resource access must get through this priority-gate, where the gate checks if the resource-requesting task has a higher priority. If the task has higher priority than the resource-blocking task, the task will be set free for the requesting task. Another solution is priority inheritance. During priority inheritance the low-priority and resource-blocking task will inherit the priority from the high-priority and resource-requesting task. This behavior makes it impossible for the medium-priority task to interrupt the schedule and to steal the high-priority task's time slice. Figure 4.3 gives a good visual representation of the problem. On the top, the graphic shows the priority inversion problem. At the bottom is the solution for the problem: priority inheritance. The line in the upper right corner symbolizes the exclusive resource access by the task at the bottom with priority level 1. Between time slice 2 and 3 a task with higher priority tries to access the resource, but gets denied (visualized by a line crossing from the bottom left to the top right corner). At time slice 5 a task with the priority level 2 gets scheduled. The priority level is higher than priority of the resource-blocking task, hence the resource-blocking task is interrupt. The high-priority task still waits for execution and has to wait until time slice 12. With priority inheritance (on the bottom) the task with priority 1 will inherit the priority from the resource-requesting high-priority task. Due to this behavior the resource blocking task has a higher priority than the medium-priority task, thus the former low-priority task can set the resource free as fast as possible and the high-priority task can execute in time slice 7, leading to a faster execution time for the high-priority task. The medium-priority task gets executed after the high-priority task, instead of before like in the upper part of the picture.

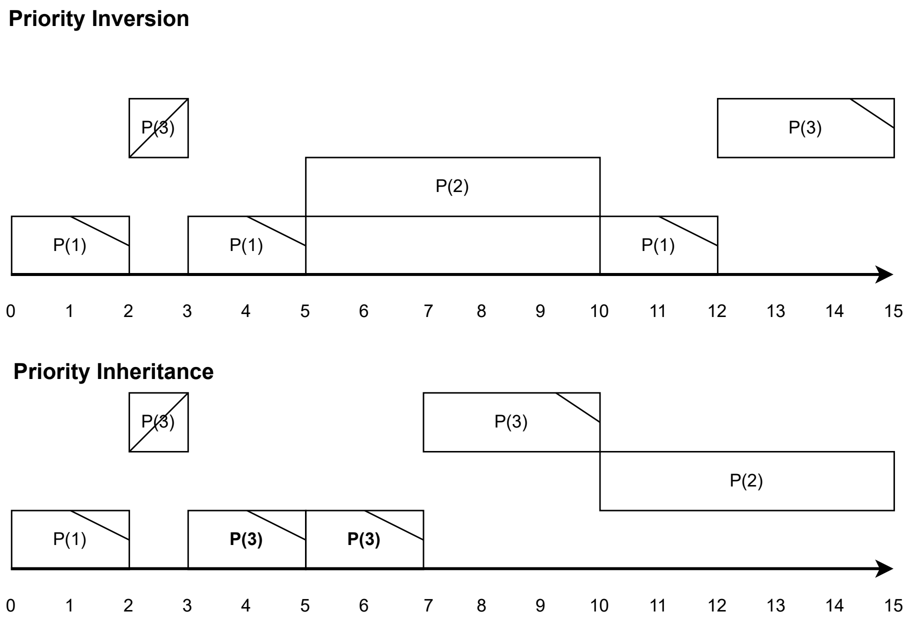


Figure 4.3: Priority inversion problem and its solution: Priority inheritance[4]

Choosing the right scheduling mechanism solves a few of the problems with real-time operating systems. Additionally, other work is needed. The underlying hardware has to meet the right requirements for power management, preemtable hardware interrupts or latency improvements. This is why the real-time patched Linux kernel disables slower CPU instruction sets[4].

4.3 Containers and Separation Kernels

Achieving all requirements for real-time operating systems is not easy. The most prominent tool in the aviation industry for satisfying the high standards is the separation kernel. Separation kernels are a combination of hardware and software for running multiple applications with real-time capabilities and more importantly without interference[29]. Interference protection can be summarized in the following three isolation practices[7]:

Fault Isolation a fault in one application should not infect other applications.

Spatial Isolation when applications are loaded into the RAM, these applications must execute within independent address spaces.

Temporal Isolation an application's real-time behavior should be correct and not be interfered by other applications.

Fault Isolation can be prevented fairly easy via Linux namespaces. Linux namespaces can isolate containers from each other. The *pid* namespace can be used to isolate processes from each other. Using this namespace will prevent managing processes outside of the container. Network access can be controlled through the *net* namespace, allowing an isolation of network layer through kernel network filters, for instance the Extended Berkeley Packet Filter (eBPF). The *ipc* namespace gives the opportunity to isolate inter-process communication. Users in containers can be controlled with the *user* namespace. Via the *user* namespace it is possible to map the user IDs in the container to a higher layer onto the host system. For example, a container could be given the full user ID set of 65535 user IDs including the root (the administrator user), but this set would then be remapped onto an extended area of user IDs on the host, such as 100000-165535. The same happens with the process IDs in the *pid* namespace. The *mnt* namespace is used for controlling mount points inside the container, perfect for data sharing, and the *uts* namespace allows setting different host names for each container. Namespaces are not the only instrument for satisfying the isolation requirements of aviation. cgroups provide a hierarchical isolation and grouping of containers. A cgroup allows to allocate, prioritize and limit memory and processor resources. However, cgroups are not capable of isolating applications in different memory pages or different processor cores. Also, spatial isolation through RAM may be not enough, it would be useful to provide CPU pinning, too. CPU pinning allows giving a process isolated access to a CPU core. This way it is possible to run applications with different priority or criticality levels on different processor cores, enabling stricter spatial isolation. The result is a mixed-criticality system with isolated CPU access. Interestingly, there are solutions for both, RAM and CPU pinning. Kubernetes supports guaranteed memory allocation for pods in its *Guaranteed* QoS class. Sadly, this solution does not provide real memory isolation, it mostly serves allocation purposes on Non-Uniform Memory Access (NUMA) architectures[142]. NUMA refers to a computing architecture, where each CPU has its own local memory for faster memory access. The counterpart for this computing architecture is called Uniform Memory Access (UMA), where multiple CPU cores access the same memory. Due to the NUMA architecture it is possible that processes are scheduled on CPU cores and remote memory, instead of the faster local memory. The Kubernetes memory manager addresses this problem by pinning the process on the local memory of the scheduled CPU. Another feature of the memory manager of Kubernetes is the support for huge memory pages (hugepages). Memory pages are chunks of data allocated by the CPU, usually in 4 kilobyte chunks, because this is the limit that all CPU vendors can handle. Hugepages provide a bigger chunk size and therefore faster memory access for Central Processing Units supporting hugepages[56]. Kubernetes solves the memory allocation problem for NUMA architectures, but what about the memory isolation problem? The Linux kernel supports memory isolation on memory page layer since version 5.14 via the new *memfd_secret* system call[79]. *memfd_secret* has its origin in research by the International Business Machines Corporation (IBM)[112][26]. According to IBM, the idea's core concept is the utilization of the hardware Memory Management Unit (MMU) for restricting access to specific memory pages. The MMU is responsible for translating virtual memmory addresses to physical memory addresses via dividing them into these memory pages[143]. Figure 4.4 depicts different memory pages. Processes running in user mode have a restricted permission set. The user page table on the left in Figure 4.4 depicts this scenario. Kernel space is not accessible on a user page table. Every interaction between a process in user mode and the kernel space have to go through the kernel entry area. The kernel entry area provides entrypoints for system calls into the Linux kernel. The right graphic shows a kernel page table with permitted access to the kernel space. With the new *memfd_secret* system call it is now possible to store memory directly in kernel space and therefore hidden for other processes running in user mode.

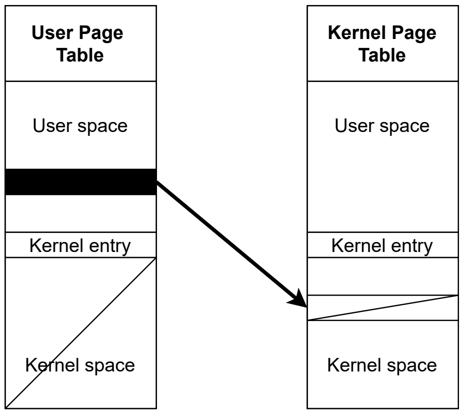


Figure 4.4: Exlusive memory mapping in the Linux kernel with Kernel version 5.14 upwards[112]

The above solution allows isolating process memory from each other, but what about CPU pinning for running processes exclusively on a dedicated CPU isolated from other processes? Pinning a process to a single CPU has multiple advantages for time-sensitive systems. First of all, if a process is not scheduled to a dedicated CPU the normal kernel scheduler takes over and schedules the process for best multi CPU utilization and multi-tasking. This behavior is non-deterministic, violates the constraints for real-time systems and is completely unsuitable for latency-sensitive workloads. Scheduling a time-sensitive process on a dedicated CPU fixes this problem. The container orchestration engine Kubernetes tries to address this problem with its CPU manager, based on research at Intel[38]. Linux distributions have three kinds of CPU resource control: shares, quotas and CPU affinity. Shares refers to the share of CPU time on a system (handled by the *requests* statement in Kubernetes) and quotas enable the system to set a hard cap of CPU time over a period (handled by the *limits* statement in Kubernetes)[38]. CPU affinity pins proccesses to a given set of Central Processing Units. The requirement for CPU pinning is the use of whole CPU core numbers, for instance 1000m for 1 CPU core. Listing 3.8 from an earlier chapter shows an example pod definition in Kubernetes with the QoS class *Guaranteed*, but a CPU share and quota of 100m. 100m is not a full CPU core, thus this pod is scheduled on a random CPU with low utilization for archieving better overall performance. With setting the value 100m to 1000m Kubernetes would change the pod's policy to *static* and would place the pod on its own CPU core. If another payload is already running on this CPU, the payload will be re-scheduled on a different CPU. Hardware-unawareness is one known limitation of the CPU manager, because Kubernetes does not know the exact CPU layout. This could lead to situations, where a pod gets scheduled on a CPU that is far more away from a time-sensitive hardware device (for example a sensor) than another free CPU. Figure 4.5 shows two Kubernetes clusters with one node each and four CPU cores. On the left cluster it is clearly visible that the scheduled pod runs processes on all four Central Processing Units. The right graphic shows the same cluster, but with with the cpu manager policy set to *static*. All processes inside of the pod run in the same CPU core (visualized by the green color). The pictures were created with the *cpuset-visualizer* project by Connor Doyle, one of the Intel researchers responsible for the cpu manager at Kubernetes[28]. Figure 4.5 proves that allocating processes on one CPU core is possible, but what about true isolation? For true isolation the Linux kernel offers a boot parameter called *isolcpus*. The boot parameter *isolcpus* prevents the Linux kernel from automatically scheduling processes on a set of Central Processing Units. With this boot parameter it is possible to deterministically isolate a CPU and prevent accidentally scheduling processes on it. This feature is not yet supported in the vanilla Kubernetes orchestration engine. Intel is actively working on a new CPU manager that makes use of the *isolcpus* kernel parameter. Intel states in their report, that the new CPU manager boosts the performance significantly in noisy neighbor scenarios and achieves a consistent performance in terms of throughput and latency[11]. Noisy neighbor scenarios occur when a neighbor CPU core has a very high utilization, leading to a performance regression of other cores.

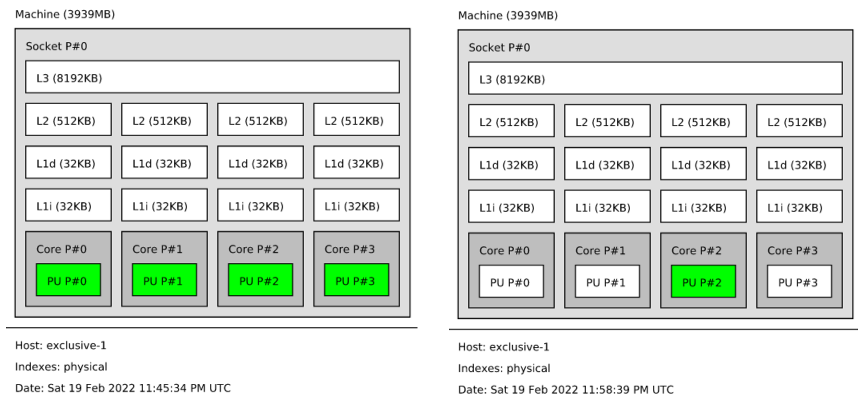


Figure 4.5: Two Kubernetes clusters with one node each and four Central Processing Units. The right node has its CPU manager policy set to *static*

Temporal isolation can be achieved by the real-time capabilities explained prior.

# Chapter 5

Test Scenarios

5.1 Introduction to the test environment

In this part of the thesis, we are going to setup a minimal test environment for evaluating the capabilities of the Kubernetes orchestration engine. As first step, we have to download Minikube. Minikube is a program written in Go for creating local test environments with Kubernetes[89]. Kubernetes installations via Minikube have been designed for testing purposes or local development only. For installing Kubernetes on the edge, in production environments, a Kubernetes distribution such like KubeEdge[63] is the better choice. Furthermore, this test scenario will not cover the Linux real-time kernel capabilities, due to space and time constraints of this thesis. There is plenty of research suggesting that the Linux real-time kernel is capable of running time-sensitive workloads with containers[116][19][47][129][42][135][20][21]. Instead, this chapter of the thesis will focus on the two main isolation criterias fault isolation and spatial isolation[7]. As first step it is necessary to install Minikube. The Minikube website provides an installation guide for Minikube on different operating systems (Linux, MacOS, Windows) and different architectures (x86-64, arm64, armv7, ppc64, s390x)[90]. Depending on the operating system a hypervisor is needed, because some operating systems like Windows or MacOS are lacking native container support. Throughout this thesis we will always use the virtual machine hypervisor, because it let us modify the Central Processing Units and RAM. Using the virtual machine does not mean that the following scenario is not possible on hardware directly. Every step in this chapter can execute on every Kubernetes installation. This is one of the core advantages of Kubernetes. Kubernetes provides an abstraction layer that makes it very convenient for production and development environments. If Minikube has been installed successfully, it is possible to invoke Minikube via the command line. The command *minikube –help* offers a first overview over Minikube's commands. Help for the subcommands will appear with the *minikube <subcommand> –help* command, for instance *minikube start –help*. Minikube provides an internal installer for the Kubectl command line tool via *minikube kubectl*. Kubectl is the native Kubernetes client for interacting with the Kubernetes cluster. Minikube only installs the cluster and manages it infrastructure. The official Kubernetes documentation maintains a Kubectl cheat sheet[62]. Figure 5.1 depicts the test environment setup with a single Kubernete cluster. The single cluster is worker and master/control node at the same time.

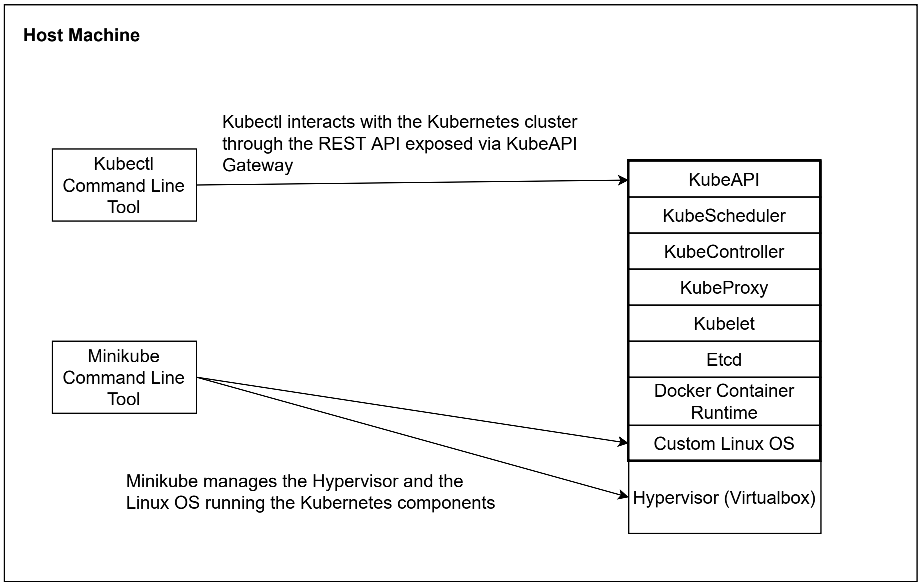

Host Machine
Kubectl Command Line Tool
Kubectl interacts with the Kubernetes cluster through the REST API exposed via KubeAPI Gateway
KubeAPI
KubeScheduler
KubeController
KubeProxy
Kubelet
Etcd
Docker Container Runtime
Custom Linux OS
Minikube Command Line Tool
Minikube manages the Hypervisor and the Linux OS running the Kubernetes components
Hypervisor (Virtualbox)

Figure 5.1: Overview over the test environment with a host system, Minikube and Kubectl command line tools and the hypervisor running a custom Linux operating system with Kubernetes

5.2 Deploying the first pod

In this section of the thesis we will deploy our first Kubernetes pod. A pod consists of one container with a *pause* process for managing the pod and any additional payload. The Figure 3.3 in an earlier chapter compares a pod and a container. First of all we need to start the Minikube instance via the Minikube command line tool. The following command requires a host system with more than 4 CPU and more than 4GB of RAM: *minikube start –cpus 4 –memory 4096 –vm*. Additional flags modify the virtual machine for the Kubernetes test environment: *–vm* sets up a virtual machine with the Virtualbox hypervisor. Listing 5.1 shows a few Kubectl commands and their output. The command *minikube kubectl get nodes* gives an overview over the nodes in the cluster and shows a single node cluster with one host called *minikube* its status *Ready* and the roles *control-plane* and *master*. These two roles mean that the node is a control plane node and a worker node. For testing and local development purposes this is fine, but should not be used in production, because for a high-availability setup a cluster needs at least three dedicated control nodes and three dedicated worker nodes. The *VERSION* key also shows that Minikube has installed a Kubernetes cluster with version v1.23.1. Line 4 shows a command for listing all namespaces. The *kube-system* namespace consists of internal Kubernetes components and the default namespace is currently empty as shown in Line 11. Line 12 creates our first Kubernetes deployment with the OCI container image *cpuset-visualizer*. In Line 15 we verify that the pod starts and after maximum one minute the pod should be ready as stated by the *READY 1/1* statement in lines 20 and 21. Line 22 exposes the deployment via a Kubernetes service of type *NodePort*. *NodePorts* exposes a Kubernetes service on a free port of one of the nodes. The process in the container is listening on port 80, hence we are exposing this port via a node port to the host system. Line 27 shows that the *NodePort* 31525 has been used. With Line 28 it is possible to automatically open a browser on localhost and port 31525. Figure 5.2 shows the graphic as seen in the web browser through the command *minikube service hello-world*. Figure 5.2 shows the hostname of our pod *hello-world-646d4b4bf7-fj9ff* and that the processes in the pod are using all four CPU cores, because no CPU manager has been enabled yet.

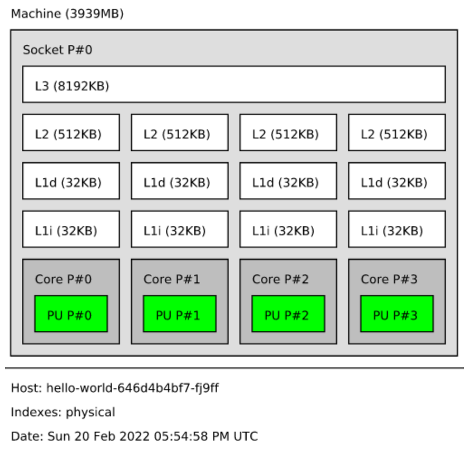


Figure 5.2: Graphical output of our first hello-world deployment with cpu-visualizer software from Intel

Listing 5.1: First steps with Minikube (the output has been slightly modified to fit on the page)

```
$ minikube kubectl get nodes
NAME       STATUS   ROLES                  VERSION
minikube   Ready    control-plane,master   v1.23.1

$ minikube kubectl get namespaces
NAME              STATUS   AGE
default           Active   65m
kube-node-lease   Active   65m
kube-public       Active   65m
kube-system       Active   65m

$ minikube kubectl get pods
No resources found in default namespace.
$ minikube kubectl create deploy hello-world -- \
  --image="quay.io/connordoyle/cpuset-visualizer"
deployment.apps/hello-world created

$ minikube kubectl get pods
NAME                               READY
pod/hello-world-646d4b4bf7-fj9ff   0/1

$ # wait a little
$ minikube kubectl get pods
NAME                               READY
pod/hello-world-646d4b4bf7-fj9ff   1/1
$ minikube kubectl expose deploy hello-world -- \
  --type=NodePort --port=80
service/hello-world exposed

$ minikube kubectl get service
NAME          TYPE       CLUSTER-IP      PORT(S)
hello-world   NodePort   10.98.232.251   80:31525/TCP

$ minikube service hello-world
```

5.3 Testing CPU and RAM allocation

Prior, we have deployed only one pod without CPU or RAM allocations. This means that our last pod has been scheduled with the Kubernetes QoS class *BestEffort*. In this section, we want to have a closer look on the resource management with cgroups. As first step, we are deleting our last deployments via *minikube kubectl delete service hello-world* and *minikube kubectl delete deploy* *hello-world*. The command *minikube kubectl get pods* should now show the status *Terminating* and the pod should be gone a few seconds later. Now, that we have a clean state we can download the *kubernetes-avionics-playground* repository from Github located at: https://github.com/shibumi/kubernetes-avionics-playground. The repository's *README.md* file gives an overview over the project and the different test scenarios. For this section, we are interested in the second test scenario located in the directory *scenario2*. Furthermore, we need to install a Minikube addon that offers us resource monitoring via the following command: *minikube addons enable metrics-server*. With the command *minikube kubectl get deploy – -n kube-system* we can verify if the *metrics-server* is up and running (we have to look for the *READY 1/1 state*). The directory *scenario2* holds a file called *limit_test.yaml*. This is the file we are looking for, for running our first test. In the first test, we want to verify, if the cgroup RAM management satisfies our expectations. Listing 5.2 shows a snippet of the *limit_test.yaml* file. Line 3 specifies the container image (a Linux Debian with release codename *buster*). Line 6 states that the container should just idle and execute a sleep command. Line 8-10 specify CPU and RAM limitations. Note that we are missing the *requests* section in this snippet. Kubernetes will automatically set the *requests* value to the same values, hence the container will have the Kubernetes QoS class *Guaranteed*. *Guaranteed* means that the pod will be scheduled on the cluster with CPU and RAM limitation and guaranteed resource allocation. Also note, that the usual *pause* container is hidden from us, because it is a Kubernetes internal container that gets deployed into every pod for management purposes. In this scenario, we will start the pod as defined in the definition *limit_test.yaml* on the cluster, then invoke a fork bomb to provoke an out-of-memory situation and then monitor what happens. A fork bomb is a parent process that creates an infinite number of child processes. Every additional child process consumes a minor amount of RAM and after a certain number of child processes the process exceeds its memory. On a physical host system a fork bomb has a devastating effect, because the host will run into a bad state (although modern defensive mechanism exist to prevent such situations). The purpose of this scenario is to test if the container will be killed before it exceeds its memory limit of 500MB. First, we are going to start the deployment via *minikube kubectl apply – -f limit_test.yaml*. Then, we will open a second terminal. In the first terminal, we are going to monitor the pod CPU and RAM consumption for the default namespace via *while true; do sleep 1; minikube kubectl top pods; done* (this command may only work on a Linux system. On a Windows host execute *minikube kubectl top pods* manually every second). In the second terminal, we start the forkbomb via jumping into the container's shell (*minikube kubectl exec deploy/hello-world – -it – /bin/bash*) and executing the fork bomb via: `:(){ :|: & };:`. In the first terminal, we should now be able to see that the memory consumption of the pod is climbing from 1MB memory consumption to over 480MB until it gets killed. Kubernetes will automatically kill the container and start a new fresh container within milliseconds. The new container will idle on 1MB memory consumption again. It is possible that all of this happens so fast, that the *metrics-server* does not show the increasing memory consumption, because the *metrics-server* is polling the monitoring API only every second. The out-of-memory situation can occur within the polling, but the second terminal should report that the container got killed via the following message: *command terminated with exit code 137*. If we lookup the Linux exit code 137, we will discover that the exit code 137 gets thrown by the out-of-memory killer on Linux. Kubernetes did indeed kill our process. The memory limitation works! Another way to verify if the Linux out-of-memory killer works, is via the command *minikube kubectl describe pod/<pod name>*. The output of this command will show the last state of Kubernetes pod as shown in Listing 5.3.

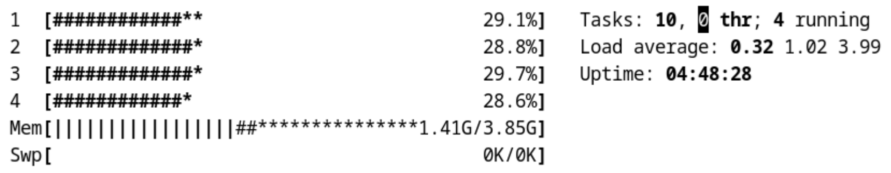
Listing 5.2: Snippet of our limit_test.yaml file with a Debian Linux container running a sleep command in a while loop and an active limit of 1 CPU and 500MB memory)

```
...
containers:
- image: debian:buster
  name: hello-world
  command: ["/bin/sh"]
  args: ["-c", "while true; do sleep 10;done"]
  resources:
    limits:
      cpu: 1000m
      memory: 500M
```

Listing 5.3: Snippet of the command *minikube kubectl describe pod/<pod name>*)

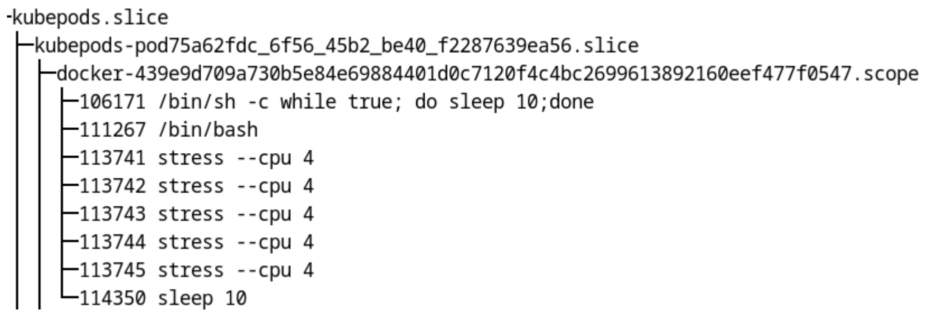
```
...
State:          Running
  Started:      Sun, 20 Feb 2022 21:40:20 +0100
Last State:     Terminated
  Reason:       OOMKilled
  Exit Code:    137
  Started:      Sun, 20 Feb 2022 21:33:48 +0100
  Finished:     Sun, 20 Feb 2022 21:40:05 +0100
Ready:          True
Restart Count:  10
Limits:
  cpu:     1
  memory:  500M
...
```

Next, we are going to test the CPU limitation. Contrary to the RAM limitations, Linux will not kill the container if it exceeds its CPU shares and quotas. Instead, Linux will just throttle the container and de-prioritize it. We need two terminals again. This time we are jumping in both terminals into the container via the following command: *minikube kubectl exec deploy/hello-world – -it – /bin/bash*. Then, we will install two tools in the container (*apt update; apt install stress htop*). *stress* is a tool for CPU stress tests and *htop* is a process viewer running in the terminal. *htop* will provide us more data than just the CPU consumption. With *htop* we are able to see which CPU cores are busy. As next step, we will invoke *htop* via the *htop* command on one of the terminals. In the other terminal, we are going to execute the stress test with CPU load tests only: *stress –cpu 4*. The parameter *–cpu 4* may seem odd at first, because the container has a limit of one CPU core, but we are doing this on purpose. The goal of this stress test is to verify if the container will exceed its CPU allocation and use more than 100% CPU. Figure 5.3 is a copy of the output of *htop*. The first important observation is that containers see all four virtual cores of the virtual machine, although the container has a CPU quota of one core. The next important observation is that, although the container has access to all four cores it cannot consume all four cores completely. Figure 5.3 shows clearly not all four cores are running on high load. The load statistic in the middle even shows an average load of 1.02, but this still throws up another question: We see four times a CPU load of 30%. Should not this be more than one virtual core? The short answer is no and there is a way to prove this. It is possible to jump on the virtual machine via the command *minikube ssh*. When inside of the SSH session (Secure Shell (SSH) is a remote administration protocol) we can invoke two commands to verify how much CPU the container consumes: *systemd-cgls*(Figure 5.4) and *systemd-cgtop*(Figure 5.5). The output of *systemd-cgls* tells us the cgroup slice name of the pod *kubepods-pod75a6..* and all 8 running processes inside of it. The output of *systemd-cgtop* on the other hand, shows us the number of processes (9, because 8 + 1 init process) and the CPU consumption that lies under the 100% threshold. The container gets indeed throttled.



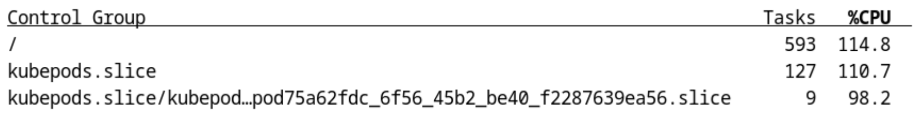
```
  1  [############**                           29.1%]   Tasks: 10, 0 thr; 4 running
  2  [#############*                           28.8%]   Load average: 0.32 1.02 3.99
  3  [#############*                           29.7%]   Uptime: 04:48:28
  4  [############*                            28.6%]
  Mem[||||||||||||||||##***************1.41G/3.85G]
  Swp[                                          0K/0K]
```

Figure 5.3: htop output in one container with a fixed CPU and RAM limit



```
-kubepods.slice
 ├─kubepods-pod75a62fdc_6f56_45b2_be40_f2287639ea56.slice
 │ ├─docker-439e9d709a730b5e84e69884401d0c7120f4c4bc2699613892160eef477f0547.scope
 │ │ ├─106171 /bin/sh -c while true; do sleep 10;done
 │ │ ├─111267 /bin/bash
 │ │ ├─113741 stress --cpu 4
 │ │ ├─113742 stress --cpu 4
 │ │ ├─113743 stress --cpu 4
 │ │ ├─113744 stress --cpu 4
 │ │ ├─113745 stress --cpu 4
 │ │ └─114350 sleep 10
```

Figure 5.4: Output of systemd-cgls showing our cgroup slice with the running processes in the container



```
Control Group                                                          Tasks   %CPU
/                                                                        593  114.8
kubepods.slice                                                           127  110.7
kubepods.slice/kubepod…pod75a62fdc_6f56_45b2_be40_f2287639ea56.slice       9   98.2
```

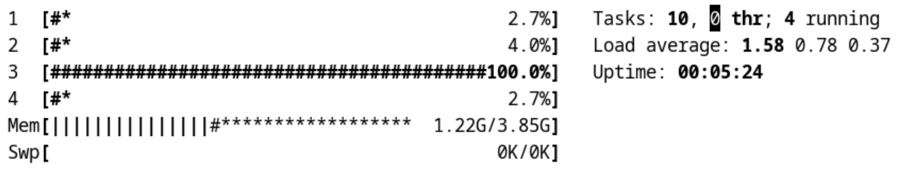
Figure 5.5: Output of systemd-cgtop showing the real CPU consumption of the container that is under the threshold of 100%

As last test for this section, we try to deploy another payload on our cluster. This time, the CPU and RAM limits will be much higher than the remaining resources. One deployment with QoS class *Guaranteed* runs already on the cluster. What will happen if we try to schedule another deployment with 4 Central Processing Units and 8000MB memory? The *kubernetes-avionics-playground* has another deployment file named *too_ much.yaml*. When deploying it (*minikube kubectl apply – -f too_ much.yaml*) and verify the deployment (*minikube kubectl get pods*) it becomes visible that the status is on *Pending* and 0/1 containers are ready. *minikube kubectl get events* reports that scheduling failed, because of insufficient CPU and insufficient memory. It did work! Kubernetes prevented the deployment, because of insufficient resources.

5.4 Testing the CPU manager

Prior this chapter had a look on spatial isolation with a high focus on resource allocation. In this section, we will investigate the spatial isolation with focus on CPU isolation. For this scenario, we have to delete the minikube instance (*minikube delete*) and restart it with the following parameters:
minikube start –cpus 4 –memory 4096 –vm
–extra-config=kubelet.cpu-manager-policy=static
–extra-config=kubelet.kube-reserved=cpu=1
–extra-config=kubelet.system-reserved=cpu=1.
There are three new flags behind the command. These flags are configuration

flags for the Kubelet, the Kubelet is the Kubernetes agent running on every Kubernetes control and worker node. With the first flag we set the CPU manager policy to *static*. The other two remaining flags are needed, because when the CPU manager's policy is set to *static* we have to configure a guaranteed set of CPU and memory for the Kubernetes components. Now, we replay the commands from the previous section. First, we deploy the *limit_ test.yaml*, then we install the stress testing tool and *htop* again. Second, we invoke the stress testing tool and *htop*. Additionally, we will deploy the *cpu-visualizer* from the first section of this chapter: *minikube kubectl – -f cpu_ visualizer.yaml*. Figure 5.6 and Figure 5.7 show the results of the third scenario. This time, *htop* shows 100% consumption on only one CPU core with a stress test over 4 cores. The CPU visualizer confirms the *htop* output.



```
1   [#*                                          2.7%]   Tasks: 10, 0 thr; 4 running
2   [#*                                          4.0%]   Load average: 1.58 0.78 0.37
3   [##########################################100.0%]   Uptime: 00:05:24
4   [#*                                          2.7%]
Mem[|||||||||||||||#******************    1.22G/3.85G]
Swp[                                            0K/0K]
```

Figure 5.6: htop output with CPU manager policy set to static

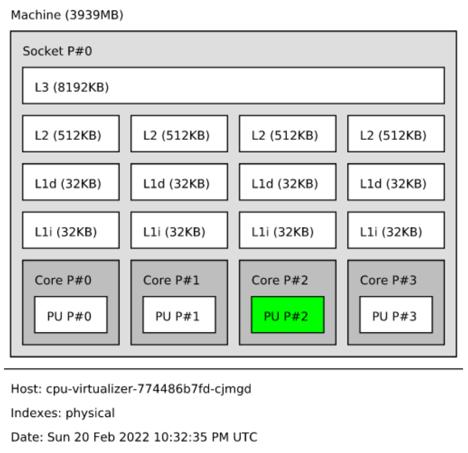


Figure 5.7: CPU visualizer output showing only one CPU is in use by the container

5.5 Testing network isolation

This short section will cover network isolation. Network isolation helps with fault isolation, because it prevents other services from affecting services over the network, for example accidentally creating too much network traffic. The directory *scenario4* covers this section in the *kubernetes-avionics-playground* repository. First, we have to delete the current minikube instance again *minikube delete* and re-create it with a custom Container Network Interface (CNI) plugin called *calico*: *minikube start –cpus 4 –memory 4096 –vm –cni=calico. Calico* is needed, because the default KubeProxy components provide no network isolation in form of network policies. With the new cluster up and running we can deploy one test server container running the web server *Nginx* and one client container running the HTTP tool *cURL*. *cURL* allows us to invoke HTTP requests on the command line. *minikube kubectl apply – -f deployments.yaml* starts both deployments. Next, we verify that the client container can reach the server container via executing *cURL*: *minikube kubectl exec deploy/client – – curl -I –silent 10.244.120.67* (the server container's IP address is different on every cluster. The command *minikube kubectl get pods – -o wide* prints the IP addresses for all pods). Listing 5.4 depicts our success. The HTTP request has been sent successfully and the server container answered it, hence there is no network isolation in place yet. In the next step, we activate a network policy as described in Listing 5.6. Line 6 has an empty pod selector, this means that the network policy will apply to all pods. Line 7-9 describes the policy types: ingress and egress. Line 10 sets the ingress policy to an empty list, blocking all incoming traffic. Lines 11-19 defines an egress rule for all namespaces to the pod that matches the labels *k8s-app: kube-dns* on port 53 and protocol UDP, allowing only DNS traffic. Listing 5.5 reports the result of the same command with network policies enabled, showing that the command fails, because the connection is running into a timeout. The network isolation works.

Listing 5.4: Snippet of the command *minikube kubectl exec deploy/client – – curl -I –silent 10.244.120.67* with network policies disabled

```
HTTP/1.1 200 OK
Server: nginx/1.21.6
Date: Sun, 20 Feb 2022 23:21:44 GMT
Content-Type: text/html
Content-Length: 615
Last-Modified: Tue, 25 Jan 2022 15:03:52 GMT
Connection: keep-alive
ETag: "61f01158-267"
Accept-Ranges: bytes
```

Listing 5.5: Snippet of the command *minikube kubectl exec deploy/client – – curl -I –silent 10.244.120.67* with network policies enabled

```
curl: (7) Failed connect to 10.244.120.67:80
Connection timed out
command terminated with exit code 7
```

Listing 5.6: Kubernetes network policy that only allows DNS traffic

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: prevent-all-traffic-except-dns
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
  ingress: []
  egress:
    - to:
        - namespaceSelector: {}
          podSelector:
            matchLabels:
              k8s-app: kube-dns
      ports:
        - port: 53
          protocol: UDP
```

Chapter 6

Conclusion

6.1 Summary

This thesis has started with the question if it is possible to apply state of the art practices from the Cloud-Native ecosystem to the aviation industry. chapter 2 gave a detailed introduction to the requirements of software systems running in the aviation industry. It started with an introduction to federated avionics and its problems with increasing cable lengths and energy consumption. Although, federation avionics has been proposed as possible solution for the problems of federated avionics, some problems remain and there is still room for further improvement through new technologies. Heavy virtual machine payloads turned out to be a good solution for isolation, but they require hardware virtualization and unnecessary layers of complexity. chapter 3 brought the necessary overview over modern containers, their orchestration, their standards, and software supply chain security. It provides a detailed view on the Cloud-Native ecosystem and highlights various Kubernetes API functionalities. Furthermore, it offers an introduction to container interfaces and standards such as OCI. chapter 4 connected both domains with each other, featured important projects in the real-time Linux domain, and proposed solutions for separation kernel implementations through containers. Intensive research and digging through many research papers, brought papers to attention that offer high value for the aviation industry, because they tackle current problems of the industry and suggests working and implementable solutions. The section containers and seperation kernels addressed the requirements of separation kernels and their isolation practices and connects them with container technologies that are up to date and either work in progress or finished just a few months ago. chapter 5 demonstrated Cloud-Native solutions for certain scenarios with focus on spatial and fault isolation. Sadly, this chapter provided no more detailed work in real-time containers and modern Linux kernel features like *memfd_secret* or the new CPU manager by Intel. Altogether, this thesis provides the foundation for future research in this area and a good entrypoint for the search for more connection points between

Cloud-Native and avionics.

6.2 Limitations

Even though this thesis provides a detailed overview over Kubernetes, avionics and modern container technologies. This thesis has a few weak spots. First of all, this thesis is far away from presenting a full overview to Kubernetes. Kubernetes is much larger than introduced in this thesis and grows rapidly due to massive contributions by big global players like Google, Intel, IBM, Microsoft or Amazon. Kubernetes offers much more API functionality than described in this thesis and has numerous sub projects extending Kubernetes or extending the Cloud-Native ecosystem in general. Figure 3.1 gives a feeling for the countless number of services and software projects related to this technology. It is possible that this thesis will not be up to date in the next years, due to Kubernetes' fast development cycles. Secondly, this thesis is missing depth in certain areas. Providing the necessary depth would have smashed the time and place constraints of this thesis, but this also means that there is plenty of more research possible and more to discover and learn. Especially interesting are the real-time capabilities of the Linux kernel in combination with containers. Companies like Intel seem to have an increasing interest in this area and are addressing similar problems to the isolation problems in the avionic industry.

6.3 Implications

The implications of this work can be summarized as follows. There is indeed an overlapping between avionics and the cloud computing industry. Containers have proven to be a good next step for running payloads in clouds and beyond. Although virtual machines are superior in isolation, kernel features in modern operating systems are catching up. Many of the discussed kernel features in this thesis have been under the radar for years and many other features are so new that the aviation industry may not discover them for the next few years. This thesis wants to change this with providing a good foundation for further research, development and entrypoints to a fully new ecosystem. The Wind River company understands this challenges and chances and seem to act aggressively in this market. Other companies have to follow this trend and have to evaluate containers for their time-sensitive application areas. Containers may be not the ideal solution for all DO-178C IDAL levels, but they can play a big part in future avionics.

Listings

List of Figures

Bibliography

[1] *About RTAI*. Department of Aerospace Sciences and Technologies Milan, Italy. URL: `https://www.rtai.org/?About_RTAI` (visited on 02/05/2022).

[2] *About the Open Container Initiative*. URL: `https://opencontainers.org/about/overview/` (visited on 11/01/2021).

[3] SAE ARP4754A. "Guidelines for development of civil aircraft and systems". In: *SAE International* (2010).

[4] Stefan Assmann. *Einführung in Real-Time Linux Preempt-RT*. Red Hat. URL: `https://chemnitzer.linux-tage.de/2010/vortraege/shortpaper/517-slides.pdf` (visited on 02/12/2022).

[5] Tjerk Bijlsma et al. "A distributed safety mechanism using middleware and hypervisors for autonomous vehicles". In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020, pp. 1175–1180.

[6] Randy Black and Mitch Fletcher. "Open systems architecture-both boon and bane". In: *2006 ieee/aiaa 25TH Digital Avionics Systems Conference*. IEEE. 2006, pp. 1–7.

[7] Jan Bredereke. "A survey of time and space partitioning for space avionics". In: (2017).

[8] Eric Brewer. "Spanner, truetime and the cap theorem". In: (2017).

[9] Daniel Bristot de Oliveira. *Deadline scheduling part 1 — overview and theory*. URL: `https://lwn.net/Articles/743740/` (visited on 02/12/2022).

[10] Ben Brosgol and Cyrille Comar. *DO-178C: A new standard for software safety certification*. Tech. rep. ADA CORE TECHNOLOGIES NEW YORK NY, 2010.

[11] Philip Brownlow and Dave Cremins. *CPU Management - CPU Pinning and Isolation in Kubernetes* Technology Guide*. Intel. 2021. URL: `https://builders.intel.com/docs/networkbuilders/cpu-pin-and-isolation-in-kubernetes-app-note.pdf` (visited on 02/19/2022).

[12] Brendan Burns et al. "Borg, omega, and kubernetes". In: *Communications of the ACM* 59.5 (2016), pp. 50–57.

[13] *Byte Magazine Volume 08 Number 10 - UNIX*. URL: `https://archive.org/details/byte-magazine-1983-10/1983_10_BYTE_08-10_UNIX?view=theater#page/n133/mode/2up` (visited on 10/03/2021).

[14] *Calico Project Website*. URL: `https://www.tigera.io/project-calico/` (visited on 12/23/2021).

[15] Justin Cappos et al. "A look in the mirror: Attacks on package managers". In: *Proceedings of the 15th ACM conference on Computer and communications security*. 2008, pp. 565–574.

[16] Todd Carpenter et al. "ARINC 659 Scheduling: Problem Definition." In: *RTSS*. 1994, pp. 165–169.

[17] *Changes to Git commit workflow*. URL: `https://news-web.php.net/php.internals/113838` (visited on 01/08/2022).

[18] *Cilium Project Website*. URL: `https://cilium.io/` (visited on 12/23/2021).

[19] Marcello Cinque and Domenico Cotroneo. "Towards lightweight temporal and fault isolation in mixed-criticality systems with real-time containers". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE. 2018, pp. 59–60.

[20] Marcello Cinque and Gianmaria De Tommasi. "Work-in-Progress: Real-Time Containers for Large-Scale Mixed-Criticality Systems". In: *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2017, pp. 369–371.

[21] Marcello Cinque et al. "Rt-cases: Container-based virtualization for temporally separated mixed-criticality task sets". In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

[22] *Cloud Native Computing Foundation Landscape*. URL: `https://landscape.cncf.io/?fullscreen=yes` (visited on 10/09/2021).

[23] *Container Network Interface (CNI)*. URL: `https://github.com/containernetworking/cni` (visited on 12/23/2021).

[24] *Container Storage Interface (CSI)*. URL: `https://github.com/container-storage-interface/spec/blob/master/spec.md` (visited on 12/20/2021).

[25] *containerd, an industry-standard container runtime with an emphasis on simplicity, robustness and portability*. URL: `https://containerd.io/` (visited on 12/20/2021).

[26] Jonathan Corbet. *Keeping memory contents secret*. 2019. URL: `https://lwn.net/Articles/804658/` (visited on 02/19/2022).

[27] *CPU Management - CPU Pinning and Isolation in Kubernetes*. Intel. URL: `https://builders.intel.com/docs/networkbuilders/cpu-pin-and-isolation-in-kubernetes-app-note.pdf` (visited on 02/12/2022).

[28] *cpuset-visualizer*. URL: `https://github.com/ConnorDoyle/cpuset-visualizer` (visited on 02/20/2022).

[29] Alfons Crespo et al. "Mixed criticality in control systems". In: *IFAC Proceedings Volumes* 47.3 (2014), pp. 12261–12271.

[30] *cri-o, lightweight container runtime for kubernetes*. URL: `https://cri-o.io/` (visited on 12/20/2021).

[31] Sandeep Dalal and Rajender Singh Chhillar. "Case studies of most common and severe types of software system failure". In: *International Journal of Advanced Research in Computer Science and Software Engineering* 2.8 (2012).

[32] Luigi De Simone and Giovanni Mazzeo. "Isolating Real-Time Safety-Critical Embedded Systems via SGX-based Lightweight Virtualization". In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE. 2019, pp. 308–313.

[33] *Distroless Images on Github*. URL: `https://github.com/GoogleContainerTools/distroless` (visited on 10/17/2021).

[34] Abhishek Dubey et al. "Enabling strong isolation for distributed real-time applications in edge computing scenarios". In: *IEEE Aerospace and Electronic Systems Magazine* 34.7 (2019), pp. 32–45.

[35] *eport on University of Minnesota Breach-of-Trust Incident*. URL: `https://lore.kernel.org/lkml/202105051005.49BFABCE@keescook/` (visited on 01/08/2022).

[36] *etcd - a distributed, reliable key-value store for the most critical data of a distributed system*. URL: `https://etcd.io/` (visited on 12/27/2021).

[37] *Executive Order on Improving the Nation's Cybersecurity*. URL: `https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/` (visited on 01/02/2022).

[38] *Feature Highlight: CPU Manager*. Intel. 2018. URL: `https://kubernetes.io/blog/2018/07/24/feature-highlight-cpu-manager/` (visited on 02/19/2022).

[39] *Fedora CoreOS*. URL: `https://getfedora.org/en/coreos?stream=stable` (visited on 12/26/2021).

[40] Wes Felter et al. "An updated performance comparison of virtual machines and linux containers". In: *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE. 2015, pp. 171–172.

[41] *filesystem bundle specification of the runtime specification*. URL: `https://github.com/opencontainers/runtime-spec/blob/main/bundle.md` (visited on 12/20/2021).

[42] Stefano Fiori, Luca Abeni, and Tommaso Cucinotta. "RT-Kubernetes - Containerized Real-Time Cloud Computing". In: (2022).

[43] *First release of Docker.* URL: `https://web.archive.org/web/20190121004233/https://blog.docker.com/2014/06/its-here-docker-1-0/` (visited on 01/03/2021).

[44] *flannel is a network fabric for containers, designed for Kubernetes.* URL: `https://github.com/flannel-io/flannel` (visited on 12/23/2021).

[45] *Flatcar Linux.* URL: `https://www.flatcar-linux.org/` (visited on 12/26/2021).

[46] *FR-EU: Airbus develops open source toolkit for mission critical applications.* URL: `https://joinup.ec.europa.eu/collection/open-source-observatory-osor/news/fr-eu-airbus-develops-open`.

[47] Fabian Frankel and Sepehr Tayari. "Real-time Scheduling in Datacentre Clusters". In: (2021).

[48] Christian M Fuchs and Advisors Stefan Schneele. "The evolution of avionics networks from ARINC 429 to AFDX". In: *Innovative Internet Technologies and Mobile Communications (IITM), and Aerospace Networks (AN)* 65 (2012), pp. 1551–3203.

[49] Rudolf Fuchsen. "Preparing the next generation of IMA: A new technology for the scarlett program". In: *2009 IEEE/AIAA 28th Digital Avionics Systems Conference*. IEEE. 2009, 7–B.

[50] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[51] *Google Remote Procedure Calls Concept Overview.* URL: `https://github.com/grpc/grpc/blob/master/CONCEPTS.md` (visited on 12/20/2021).

[52] Jim Gruen. "Linux in space". In: *The Linux Foundation* (2012).

[53] Sanghyun Han and Hyun-Wook Jin. "Kernel-level ARINC 653 partitioning for Linux". In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. 2012, pp. 1632–1637.

[54] Jaap-Henk Hoepman and Bart Jacobs. "Increased security through open source". In: *Communications of the ACM* 50.1 (2007), pp. 79–83.

[55] Jordan K. Hubbard. *FreeBSD 4.0 Announcement.* URL: `https://www.freebsd.org/releases/4.0R/announce/` (visited on 10/03/2021).

[56] *Hugepages.* URL: `https://wiki.debian.org/Hugepages` (visited on 02/19/2022).

[57] *Introducing Container Runtime Interface (CRI) in Kubernetes.* URL: `https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/` (visited on 12/29/2021).

[58] *Introducing SLSA, an End-to-End Framework for Supply Chain Integrity.* Google. URL: `https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html` (visited on 01/08/2022).

[59] Yasemin Isik. “ARINC 629 data bus standard on aircrafts”. In: *Recent Researches in Circuits, Systems, Electronics, Control & Signal Processing* (2010), pp. 191–195.

[60] S. Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. http://www.rfc-editor.org/rfc/rfc4648.txt. RFC Editor, Oct. 2006. URL: http://www.rfc-editor.org/rfc/rfc4648.txt.

[61] Heiko Koziolek et al. “Dynamic Updates of Virtual PLCs deployed as Kubernetes Microservices”. In: *European Conference on Software Architecture*. Springer. 2021, pp. 3–19.

[62] *Kubectl Cheat Sheet*. The Linux Foundation. URL: https://kubernetes.io/docs/reference/kubectl/cheatsheet/ (visited on 02/20/2022).

[63] *KubeEdge Project Website*. The Linux Foundation. URL: https://kubeedge.io/en/ (visited on 02/20/2022).

[64] *Kubernetes Code on Github showing that the sandbox container has just a idle process*. URL: https://github.com/kubernetes/kubernetes/blob/b8ce285a03a7e9d59e94712248ab96c54a3216a3/build/pause/linux/pause.c#L65 (visited on 12/29/2021).

[65] *Kubernetes Components*. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 12/28/2021).

[66] *Kubernetes DaemonSet*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/ (visited on 12/30/2021).

[67] *Kubernetes Docker Deprecation*. URL: https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/ (visited on 12/26/2021).

[68] *Kubernetes first commit*. URL: https://github.com/kubernetes/kubernetes/commit/2c4b3a562ce34cddc3f8218a2c4d11c7310e6d56 (visited on 12/25/2021).

[69] *Kubernetes Github Webpage*. URL: https://github.com/kubernetes/kubernetes (visited on 12/24/2021).

[70] *Kubernetes Pod Documentation*. URL: https://kubernetes.io/docs/concepts/workloads/pods/ (visited on 12/25/2021).

[71] *Kubernetes QOS Classes*. URL: https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/ (visited on 12/30/2021).

[72] *Kubernetes Service*. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on 01/02/2022).

[73] *Kubernetes StatefulSets*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/ (visited on 12/30/2021).

[74] *Kubernetes Website*. URL: https://kubernetes.io/ (visited on 12/24/2021).

[75] *KubeVip project documentation*. URL: https://kube-vip.io/ (visited on 02/21/2022).

[76] SJ Lemke. *A Comparative Evaluation of the Reliability Improvement in Line Replaceable Units Warranted under the F-16 Reliability Improvement Warranty.* Tech. rep. AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF SYSTEMS and LOGISTIC S, 1985.

[77] Hannu Leppinen. "Current use of Linux in spacecraft flight software". In: *IEEE Aerospace and Electronic Systems Magazine* 32.10 (2017), pp. 4–13.

[78] Zheng Li, Qiao Li, and Huagang Xiong. "Avionics clouds: A generic scheme for future avionics systems". In: *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC).* IEEE. 2012, 6E4–1.

[79] *Linux Kernel 5.15 Changelog.* The Linux Foundation. URL: `https://cdn.kernel.org/pub/linux/kernel/v5.x/ChangeLog-5.14` (visited on 02/19/2022).

[80] *man page for Linux control groups.* URL: `https://man7.org/linux/man-pages/man7/cgroups.7.html` (visited on 02/21/2022).

[81] *man page for Linux kernel capabilities.* URL: `https://man7.org/linux/man-pages/man7/capabilities.7.html` (visited on 08/01/2021).

[82] *man page for Linux namespaces.* URL: `https://man7.org/linux/man-pages/man7/namespaces.7.html` (visited on 02/21/2022).

[83] M McCabe, C Baggerman, and D Verma. "Avionics architecture interface considerations between constellation vehicles". In: *2009 IEEE/AIAA 28th Digital Avionics Systems Conference.* IEEE. 2009, 1–E.

[84] Peter M Mell and Timothy Grance. *Sp 800-145. the nist definition of cloud computing.* 2011.

[85] Rebecca T Mercuri and Peter G Neumann. "Security by obscurity". In: *Communications of the ACM* 46.11 (2003), p. 160.

[86] *Metallb BGP documentation.* URL: `https://metallb.universe.tf/concepts/bgp/` (visited on 02/21/2022).

[87] *Metallb layer 2 documentation.* URL: `https://metallb.universe.tf/concepts/layer2/` (visited on 02/21/2022).

[88] *Metallb project documentation.* URL: `https://metallb.universe.tf/` (visited on 01/02/2021).

[89] *Minikube.* URL: `https://minikube.sigs.k8s.io/docs/` (visited on 12/28/2021).

[90] *Minikube Start.* URL: `https://minikube.sigs.k8s.io/docs/start/` (visited on 02/20/2022).

[91] *Namespaces compatibility list.* URL: `https://www.kernel.org/doc/html/latest/admin-guide/namespaces/compatibility-list.html` (visited on 02/21/2022).

[92] *Namespaces in operation, part 1: namespaces overview.* URL: https://lwn.net/Articles/531114/ (visited on 02/21/2022).

[93] *NASA confirms Ingenuity not vulnerable to log4j flaw.* TechRadar. URL: https://www.techradar.com/news/even-the-ingenuity-mars-helicopter-is-vulnerable-to-log4j (visited on 01/08/2022).

[94] *New Cloud Native Computing Foundation to drive alignment among container technologies.* URL: https://www.cncf.io/announcements/2015/06/21/new-cloud-native-computing-foundation-to-drive-alignment-among-container-technologies/ (visited on 10/03/2021).

[95] *Number of flights performed by the global airline industry from 2004 to 2021.* Statista. 2021. URL: https://www.statista.com/statistics/564769/airline-industry-number-of-flights/ (visited on 02/21/2022).

[96] *OCI Distribution Specification.* URL: https://github.com/opencontainers/distribution-spec/blob/main/spec.md (visited on 11/14/2021).

[97] *OCI Filesystem Layer Specification.* URL: https://github.com/opencontainers/image-spec/blob/main/layer.md (visited on 12/18/2021).

[98] *OCI Image Configuration Specification.* URL: https://github.com/opencontainers/image-spec/blob/main/config.md (visited on 12/18/2021).

[99] *OCI Image Index Specification.* URL: https://github.com/opencontainers/image-spec/blob/main/image-index.md (visited on 12/18/2021).

[100] *OCI Image Layout Specification.* URL: https://github.com/opencontainers/image-spec/blob/main/spec.md (visited on 12/18/2021).

[101] *OCI Image Manifest Specification.* URL: https://github.com/opencontainers/image-spec/blob/main/manifest.md (visited on 12/18/2021).

[102] *OCI Image Specification.* URL: https://github.com/opencontainers/image-spec/blob/main/spec.md (visited on 11/14/2021).

[103] *OCI Runtime Specification.* URL: https://github.com/opencontainers/runtime-spec/blob/main/spec.md (visited on 11/14/2021).

[104] *Official Docker documentation: Docker security.* URL: https://docs.docker.com/engine/security/ (visited on 02/21/2022).

[105] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14).* 2014, pp. 305–319.

[106] Diego Ongaro and John Ousterhout. *In search of an understandable consensus algorithm (extended version).* 2013.

[107] Matjaž Pančur and Mojca Ciglarič. "Impact of test-driven development on productivity, code and tests: A controlled experiment". In: *Information and Software Technology* 53.6 (2011), pp. 557–573.

[108] M Pernpeintner. “You can’t fix the unknown. Incident response and vulnerability management in automotive”. In: *VDI-Berichte* 2358 (2019).

[109] David Plumer. *An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware*. RFC 826. Nov. 1982. DOI: `10.17487/RFC0826`. URL: `https://rfc-editor.org/rfc/rfc826.txt`.

[110] *Post-Mortem / Root Cause Analysis (April 2021)*. Codecov. URL: `https://about.codecov.io/apr-2021-post-mortem/` (visited on 01/09/2022).

[111] Paul J Prisaznuk. “Integrated modular avionics”. In: *Proceedings of the IEEE 1992 National Aerospace and Electronics Conference@ m_ NAECON 1992*. IEEE. 1992, pp. 39–45.

[112] Mike Rapoport and James Bottomley. *Address Space Isolation in the Linux Kernel*. IBM. URL: `https://archive.fosdem.org/2020/schedule/event/kernel_address_space_isolation/attachments/slides/3889/export/events/attachments/kernel_address_space_isolation/slides/3889/Address_Space_Isolation_in_the_Linux_Kernel.pdf` (visited on 02/19/2022).

[113] Emily Ratliff. *Establishing Correspondence Between an Application and its Source Code*. URL: `https://www.securityweek.com/establishing-correspondence-between-application-and-its-source-code` (visited on 01/23/2022).

[114] *Real-time operating system for microcontrollers*. FreeRTOS. 2021. URL: `https://www.freertos.org/index.html` (visited on 02/21/2022).

[115] *Request For Comments, Wikipedia article*. URL: `https://en.wikipedia.org/wiki/Request_for_Comments` (visited on 12/31/2021).

[116] *RTAI - Real Time Application Interface Official Website*. Department of Aerospace Sciences and Technologies Milan, Italy. URL: `https://www.rtai.org/` (visited on 02/05/2022).

[117] *Running Virtual Machines in Kubernetes*. URL: `https://kubevirt.io/` (visited on 12/25/2021).

[118] *sched - overview of CPU scheduling*. The Linux Foundation. URL: `https://man7.org/linux/man-pages/man7/sched.7.html` (visited on 02/12/2022).

[119] *Scheduling - Policy and Priority*. The Linux Foundation. URL: `https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/sched_policy_prio/start` (visited on 02/12/2022).

[120] Zach Schneider. *event-stream vulnerability explained*. URL: `https://schneider.dev/blog/event-stream-vulnerability-explained/` (visited on 01/09/2022).

[121] Malte Schwarzkopf et al. “Omega: flexible, scalable schedulers for large compute clusters”. In: *Proceedings of the 8th ACM European Conference on Computer Systems.* 2013, pp. 351–364.

[122] Kyoung-Taek Seo et al. “Performance comparison analysis of linux container and virtual machine for building cloud”. In: *Advanced Science and Technology Letters* 66.105-111 (2014), p. 2.

[123] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. “Priority inheritance protocols: An approach to real-time synchronization”. In: *IEEE Transactions on computers* 39.9 (1990), pp. 1175–1185.

[124] Ax Sharma. *Damaging Linux and Mac Malware Bundled within Browserify npm Brandjack Attempt.* URL: `https://blog.sonatype.com/damaging-linux-mac-malware-bundled-within-browserify-npm-brandjack-attempt` (visited on 01/09/2022).

[125] Balbir Singh and Vaidyanathan Srinivasan. “Containers: Challenges with the memory resource controller and its performance”. In: *Ottawa Linux Symposium (OLS).* Citeseer. 2007, p. 209.

[126] *SLSA Levels.* Linux Foundation. URL: `https://slsa.dev/spec/v0.1/levels` (visited on 01/15/2022).

[127] *SLSA Project website.* Linux Foundation. URL: `https://slsa.dev/` (visited on 01/15/2022).

[128] *SLSA Requirements.* Linux Foundation. URL: `https://slsa.dev/spec/v0.1/requirements` (visited on 01/15/2022).

[129] Václav Struhár et al. “REACT: Enabling Real-Time Container Orchestration”. In: *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA).* IEEE. 2021, pp. 1–8.

[130] Syeda Sulaiman. *Top Programming Languages in Aerospace.* Aversan. URL: `https://www.aversan.com/top-programming-languages-in-aerospace/`.

[131] *SUNSPOT: An Implant in the Build Process.* URL: `https://www.crowdstrike.com/blog/sunspot-malware-technical-analysis/` (visited on 01/09/2022).

[132] *syscalls(2) - Linux manual page.* The Linux Foundation. URL: `https://man7.org/linux/man-pages/man2/syscalls.2.html` (visited on 02/12/2022).

[133] *Systemd.* URL: `https://systemd.io/` (visited on 12/26/2021).

[134] Andrew Tanenbaum. *Modern operating systems.* Pearson Education, Inc., 2009.

[135] *Technical basics: Important aspects for real time.* Linux Foundation. URL: `https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/start` (visited on 02/12/2022).

[136] *The Extended Berkeley Packet Filter.* URL: https://ebpf.io/ (visited on 12/25/2021).

[137] *The Linux Foundation - Real-Time Linux.* The Linux Foundation. 2021. URL: https://wiki.linuxfoundation.org/realtime/start (visited on 02/21/2022).

[138] *The US is readying sanctions against Russia over the SolarWinds cyber attack. Here's a simple explanation of how the massive hack happened and why it's such a big deal.* URL: https://www.businessinsider.com/solarwinds-hack-explained-government-agencies-cyber-security-2020-12 (visited on 01/02/2022).

[139] *U-2 Federal Lab achieves flight with Kubernetes.* Air Combat Command Public Affairs. 2020. URL: https://www.af.mil/News/Article-Display/Article/2375297/u-2-federal-lab-achieves-flight-with-kubernetes/.

[140] *Usage statistics of PHP for websites.* URL: https://w3techs.com/technologies/details/pl-php (visited on 01/08/2022).

[141] *Using mlock to avoid page I/O.* Red Hat. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/using_mlock_to_avoid_page_io (visited on 02/12/2022).

[142] *Utilizing the NUMA-aware Memory Manager.* URL: https://kubernetes.io/docs/tasks/administer-cluster/memory-manager/ (visited on 02/19/2022).

[143] Frank Uyeda. *CSE 120: Principles of Operating Systems.* UC San Diego. 2009. URL: https://cseweb.ucsd.edu/classes/su09/cse120/lectures/Lecture7.pdf (visited on 02/19/2022).

[144] Steven H VanderLeest. "ARINC 653 hypervisor". In: *29th Digital Avionics Systems Conference.* IEEE. 2010, 5–E.

[145] Ivan Velichko. *Containers vs. Pods - Taking a Deeper Look.* URL: https://iximiuz.com/en/posts/containers-vs-pods/ (visited on 12/29/2021).

[146] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems.* 2015, pp. 1–17.

[147] Adolfo García Veytia. *What an SBOM Can Do for You.* Chainguard Inc. URL: https://blog.chainguard.dev/what-an-sbom-can-do-for-you/ (visited on 01/15/2022).

[148] *VxWorks 653 Platform project page.* Wind River. URL: https://www.windriver.com/products/vxworks/safety-platforms#vxworks_653 (visited on 01/23/2022).

[149] *VxWorks, the leading RTOS for the intelligent edge.* VxWorks. 2021. URL: `https://www.windriver.com/products/vxworks` (visited on 02/21/2022).

[150] Christopher B Watkins and Randy Walter. "Transitioning from federated avionics architectures to integrated modular avionics". In: *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*. IEEE. 2007, 2–A.

[151] *Webmin 1.890 Exploit - What Happened?* URL: `https://www.webmin.com/exploit.html` (visited on 01/09/2022).

[152] *Wind River VxWorks 653 platform product note.* Wind River. URL: `https://cdn.windriver.com/products/product-notes/vxworks-653-product-note/vxworks-653-product-note.pdf` (visited on 01/29/2022).

[153] *Wind River VxWorks 653 Platform Product Overview.* Wind River. URL: `https://resources.windriver.com/vxworks/vxworks-653-product-overview` (visited on 01/23/2022).

[154] Brian Witten, Carl Landwehr, and Michael Caloyannides. "Does open source improve system security?" In: *IEEE Software* 18.5 (2001), pp. 57–61.

[155] Heinz Wörn. *Echtzeitsysteme: Grundlagen, Funktionsweisen, Anwendungen.* Springer-Verlag, 2006.

[156] HG Xiong and ZH Wang. "Advanced avionics integration techniques". In: *National Defense Industry Press, Beijing, China* (2009).

Glossary

API Application Programming Interface. 12, 26, 30, 31, 35–38, 41–43, 51, 65, 71, 72

APPS ARINC plus priority-preemptive scheduling. 50

ARINC Aeronautical Radio Incorporated. 49, 50

ARINC 429 ARINC standard for a global data bus in aviation. 5

ARINC 629 ARINC standard for a global computer bus in aviation. 5

ARINC 653 ARINC standard for space and time partitioning. 5

ARP Address Resolution Protocol. 43

ARP4754 Guideline for the development of aircraft systems by SAE International. 10

BGP Border Gateway Protocol. 43

BLOB Binary Large Object. 21, 22

BPF Berkeley Packet Filter. 13

CFS Completely Fair Scheduler. 52

cgroup Linux control groups. 12, 13, 34–36, 50, 51, 56, 63

CNCF Cloud Native Computing Foundation. 16, 27

CNI Container Network Interface. 26, 27, 37, 69

CPU The CPU consists of registers for fast computation and an Algorithmic Logic Unit (ALU). 5, 10, 11, 13, 30, 35–37, 49–51, 53, 55–68, 71, 74

CRI Container Runtime Interface. 22, 26, 34

CSI Container Storage Interface. 26, 27

DAL Design Assurance Level. 9, 49

DIMA Distributed computer network airborne system. 7, 8

DNS The Domain Name System (DNS) connects every IP address with a unique fully qualified domain name. 13, 43, 69

DO-178B Certification for safety-critical software by the RTCA. 1, 47, 48

DO-178C Certification for safety-critical software by the RTCA (replaces DO-178B). 1, 48–50, 72

DO-278A Ground-Based Software Certification Guidance. 48, 49

DO-330 Software Tool Qualification Considerations. 48

DO-331 Model-Based Development and Verification Supplement to DO-178C and DO-278A. 49

DO-332 Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A. 49

DO-333 Formal Methods Supplement to DO-178C and DO-278A. 49

eBPF Extended Berkeley Packet Filter. 11, 13, 34, 56

FIFO First In, First Out. 51

FQDN Fully Qualified Domain Name. 43

GEDF Global Earliest Deadline First. 53

GID Group ID. 12, 13

gRPC Google Remote Procedure Calls. 26

HSD Horizontal Situation Display. 9

HTTP Hypertext Transfer Protocol. 35, 43, 69

HTTPS Hypertext Transfer Protocol Secure. 35, 43

I/O input and output. 3, 5, 11, 13

IaaS Infrastructure as a Service. 8

IDAL Item Development Assurance Level. 9, 49, 72

IDL Interface Definition Language. 26

IETF Internet Engineering Task Force. 41

IMA Computer network airborne system. 5–8, 49, 50

IP Internet Protocol. 12, 27, 43, 69

IPC Inter-Process Communication. 12, 13, 34

JIT just-in-time. 13

JSON JavaScript Object Notation. 35

LRM Line-Replaceable Module. 3

LRU Line-Replaceable Unit. 3–6, 74

LVM Logical Volume Manager. 26

MAC Media Access Control. 43

MMU Memory Management Unit. 57

NASA National Aeronautics and Space Administration. 45

NFS Network File System. 26

NIST United States institute for promoting innovation and industrial competitiveness. 8

NUMA Non-Uniform Memory Access. 56, 57

OCI Open Container Initiative. 20–22, 26, 27, 40, 62, 71

OSI Open Systems Interconnection model. 27

PaaS Platform as a Service. 8

PID Process ID. 12

QoS Quality of Service. 10, 35, 36, 51, 56, 58, 63, 64, 67

RAM RAM is very fast memory for temporary storing data. 5, 10, 11, 30, 35–38, 51, 52, 56, 60, 61, 63–65, 67

RBAC Role-Based Access Control. 31

REST Representational State Transfer. 35

RFC Request For Comments. 41

RPC Remote Procedure Calls. 13, 26

RTAI Real Time Application Interface. 50–52

RTCA Radio Technical Commission for Aeronautics (RTCA). 1

SaaS Software as a Service. 8

SBOM Software Bill of Materials. 46

SCM Source Code Management. 45, 47

SHA256 Secure Hash Algorithm 2. 19, 21

SLSA Supply chain Levels for Software Artifacts. 46, 47, 49

TCO Total Cost of Ownership. 49

TCP Transmission Control Protocol. 35

TTD Test Driven Development. 49

UDP User Datagram Protocol. 44, 69

UID User ID. 12, 13

UMA Uniform Memory Access. 57

US United States of America. Short: United States. 1, 2

UTS Unix Time Sharing. 12

WCET Worst-Case Execution Time. 51

WWW World Wide Web. 45

XML Extensible Markup Language. 49

YAML Yet Another Markup Language. 35, 39